COUNCIL OF THE EUROPEAN UNION

Brussels, 17 November 2010

15456/10

COPEN 236
JURINFO 49
EJUSTICE 104

NOTE

from:	General Secretariat of the Council
to:	Delegations
Subject:	ECRIS Technical Specifications - Technical Architecture

Delegations will find in the Annex the revised text of the ECRIS Technical Specifications - Technical Architecture, as agreed on at the Working Party on Cooperation in Criminal matters which met on 20 October 2010.

European Commission – DG Justice
iLICONN Consortium (Bilbomatica – Intrasoft – Unisys)

ECRIS TECHNICAL Specifications

Technical Architecture

Document Information

AUTHOR	iLICONN – Intrasoft International S.A.
OWNER	European Commission – DG Justice
ISSUE DATE	22/10/2010
VERSION	1.0
APPROVAL STATUS	Adopted

Authors

NAME	ACRONYM	ORGANISATION	ROLE
Nicholas YIALELIS	NYI	iLICONN – Intrasoft International S.A.	Manager Reviewer
Panos ATHANASIOU	PAT	iLICONN – Intrasoft International S.A.	Main Author
Ludovic COLACINO DIAS	LCO	iLICONN – Intrasoft International S.A.	Contributor Reviewer

Document History

VERSION	DATE	AUTHOR	DESCRIPTION
0.1	23/09/2010	PAT	First draft
0.2	24/09/2010	PAT	Updates in chapters 6 and onwards, examples
0.3	26/09/2010	LCO	Revision
0.4	12/10/2010	PAT	Update of all sections according to author's position on comments from Member States
0.5	14/10/2010	LCO	Revision and finalisation of document
1.0	22/10/2010	LCO	Text of version 0.5, adopted in Council by COPEN Working Party on 20-Oct-2010

TABLE OF CONTENTS

DOCUMENT

Purpose

This document is a formal product of the *ECRIS Technical Specifications* project for the European Commission – DG Justice and produced by the iLICONN Consortium.

The main purpose of this document is to describe the general technical architecture and major technical design choices upon which the detailed technical specifications of the ECRIS data exchanges are built. It provides the principles, methods, techniques, protocols and standards to be used for realising the *ECRIS Technical Specifications*.

This document assumes that the readers have a good and detailed knowledge and understanding of the following elements:

- ECRIS legal basis
- The "ECRIS Technical Specifications – Inception Report" document
- *Web Services* and in particular *service contract* design
- *XSD* and *XML* concepts

Scope

This document provides all necessary background information so as to describe the decisions in regards to the "Technical Architecture" themes. In particular, this document provides:

- the general view of the technical ECRIS system architecture and communication mechanisms
- the detailed list of technologies, standards, formats and protocols to be used for the computerised ECRIS communications
- descriptions of the design choices and technical principles to be applied for error handling, implementing transactional behaviour, designing the XSD schemas, implementing the versioning of ECRIS
- descriptions of the proposals that were considered but which have been discarded

This document does not provide any other information than what has been stated above, and in particular it does not include:

- Additional background material on the subjects discussed
- Proposals for security-related matters and the technical solutions to these (please note that such issues are being handled in a separate and specific document)
- Functional or judicial considerations; indeed this document focuses on the technical aspects of the ECRIS data exchanges rather than on the content of the messages

References

The following sources have been used as input for the elaboration of this "Technical Architecture" document:

[1] ECRIS Legal Basis – Council Framework Decision 2009/315/JHA
Council of the European Union (2009), Council Framework Decision 2009/315/JHA of 26 February 2009 on the organisation and content of the exchange of information extracted from the criminal record between Member States (OJ L 93/23 of 07.04.2009)

[2] ECRIS Legal Basis – Council Decision 2009/319/JHA
Council of the European Union (2009), Council Decision 2009/316/JHA of 6 April 2009 on the establishment of the European Criminal Records Information System (ECRIS) in application of Article 11 of Framework Decision 2009/315/JHA (OJ L 93/33 of 07.04.09)

[3] Network of Judicial Registers (NJR) – Technical References – version 1.3a (approved) of 13 March 2008

[4] Network of Judicial Registers (NJR) – Technical References – version 1.4 (draft) of 23 November 2009

[5] Network of Judicial Registers (NJR) – *XML* Listings – version 1.4 (final) of 01 July 2009

[6] NJR *WSDL* and *XML* Files v1.4.2 of 21 January 2009 (final)
"CommonTables_and_XML_rel1-4-2_20090121.zip" file containing:

- RegisterService-1.4.2.wsdl (version 1.4.2)
- common.xsd (version 1.4 of 18 December 2008)
- CommonTables-1.3.xsd (version 1.3)
- CommonTables-1.4.2.xml (version 1.4.2)
- error.xsd (version 1.4 of 02 November 2005)
- information.xsd (version 1.4 of 02 November 2005)
- notification.xsd (version 1.4 of 22 November 2005)
- receipt.xsd (version 1.4 of 02 November 2005)
- request.xsd (version 1.4 of 02 November 2005)

[7] NJR *WSDL* and *XML* Files v1.5 (draft)
- RegisterService-1.5.wsdl (draft version 1.5 of 11 August 2010)
- common.xsd (draft version 1.5 of 10 June 2010)
- CommonTables-1.5.xsd (draft version 1.5)
- CommonTables-1.5.xml (draft version 1.5.0)

– error.xsd (draft version 1.5 of 10 July 2010)
– information.xsd (draft version 1.5 of 10 July 2010)
– notification.xsd (draft version 1.5 of 10 July 2010)
– receipt.xsd (draft version 1.5 of 10 July 2010)
– request.xsd (draft version 1.5 of 10 July 2010)

[8] European Commission – DG Enterprise (2004): IDA Architecture Guidelines for Trans-European Telematics Networks for Administrations, version 7.1 of 13 February 2004 (and annexes)

[9] ECRIS Technical Specifications - Inception Report v1.02 of 22 October 2010

[10] The replies from the following Member States to the "Inception Phase Questionnaire" v0.05 (listed in alphabetical order):
Austria (AT), Belgium (BE), the Czech Republic (CZ), Estonia (EE), Finland (FI), France (FR), Germany (DE), Hungary (HU), Lithuania (LT), Luxembourg (LU), the Netherlands (NL), Poland (PL), Portugal (PT), Romania (RO), Slovakia (SK), Slovenia (SI), Spain (ES), Sweden (SE), the United Kingdom (UK)

[11] "SOA Design Patterns"
Thomas Erl - Prentice Hall/PearsonPTR (ISBN: 0136135161)

[12] "Web Service Contract Design and Versioning for SOA"
Thomas Erl, Anish Karmarkar, Priscilla Walmsley, Hugo Haas, Umit Yalcinalp, Canyang Kevin Liu, David Orchard, Andre Tost, James Pasley - Prentice Hall/PearsonPTR
(ISBN: 013613517X)

[13] "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions"
Gregor Hohpe, Bobby Wolf - Addison-Wesley Professional
(ISBN-13: 978-0321200686)

[14] "Creating flexible and extensible *XML* schemas"
Ayesha Malik, 01 Oct 2002
http://www.ibm.com/developerworks/library/x-flexschema/

[15] Guide to Versioning *XML* Languages using *XML* Schema 1.1
David Orchard (2006 W3C)
http://www.w3.org/TR/xmlschema-guide2versioning/

[16] ECRIS Technical Specifications - Technical Architecture Proposals v0.05 of 10 September 2010

[17] Comments on ECRIS Technical Architecture Proposals received from the following Member States: BE, DE, EE, EL, ES, FR, LU, RO, SE, SK, UK

[18] ECRIS Technical Specifications - Expert Sub Group Meeting Minutes and Conclusions of 22 September 2010

[19] ECRIS Technical Specifications – Glossary v1.0 of 05 October 2010

About this Document

Elaboration of this Document

This "Technical Architecture" document has been drafted by the iLICONN staff based on the following input:

- The documents listed in the references above
- The answers provided by the following Member States' central authorities to the concrete technical proposals described in the "ECRIS Technical Architecture proposals" document that has been sent out by iLICONN to all Member States' contact points on the 13th of September 2010 (listed in alphabetical order):
  Belgium (BE), the Czech Republic (CZ), France (FR), Germany (DE), Greece (GR), Luxembourg (LU), Romania (RO), Slovakia (SK), Spain (ES), Sweden (SE), the United Kingdom (UK)
- The discussions and conclusions that have been reached during the Expert Sub Group Meeting on 22nd of September 2010 with the technical experts of the following Member States: DE, EE, ES, LT, UK
- The 70 comments issued by the Member States on the previous version of this document by the 06th of October 2010.

Understanding this Document

This document comes with a "Glossary" document that provides definitions for the specific terms that are used throughout the *ECRIS Technical Specifications* project.

By convention, all words marked in italic in this document can be looked up in the "Glossary" document. The bold font is used for emphasising a specific term or part of a sentence. The underlines mark the text that has been added or modified since the last version while the strike-through marks the text that has been removed or replaced.

In case of doubts about the exact meaning of a term, please consult first the "Glossary".

Should you still have any doubts about the meaning of a specific sentence or paragraph, please do not hesitate to take direct contact with the following persons by telephone or via e-mail, at your best convenience:

Organisation: European Commission – DG Justice – Criminal Law

Name: Jaime LOPEZ-LOOSVELT

E-mail: JUST-CRIMINAL-RECORD@ec.europa.eu

Telephone: +32 (0)2.298.41.54

Organisation: iLICONN Consortium – Intrasoft International S.A.

Name: Ludovic COLACINO DIAS

E-mail: ECRIS-Specs-PM.iLICONN@intrasoft-intl.com

Mobile: +32 (0)498.30.25.55

Providing Comments

As described in the "Inception Report" document, all major deliverables produced by the iLICONN Consortium are undergoing a "Review Cycle" during which all EU Member States experts are invited to provide comments.

Since the iLICONN staff needs to collect, compare and analyse the feedback from 27 Member States on the same document – thus potentially a large number of comments – it uses a tool that allows easily extracting the comments from MS Word documents.

Therefore, for commenting this document, please apply the following guidelines:

- All comments are to be written in plain English. Comments provided in other languages cannot, unfortunately, be taken into account.
- The comments must be specific to and must relate to the text (sentence and/or paragraph) being revised.
- Please use simple wording and be as specific, concise and clear as possible in order to avoid ambiguities.
- When referring to specific terms, acronyms, abbreviations that are common in your daily jargon but that are not defined in the *Glossary* document, please define them first.
- Write your comments directly in this MS Word document, by proceeding as follows:
  - First select a word, a part of a sentence or a paragraph (this can be done for example by double-clicking on a word or by dragging your mouse over parts of the text while keeping the left mouse-button pressed).

    Attention:

Please note that a **minimum of 4 characters** must be selected in order for our commenting tool to grab the comment. Furthermore, comments on diagrams and embedded pictures are also not taken into account. In such cases, please select the caption text underneath the diagram or image.

– Once a word, part of a sentence or paragraph has been selected, insert an MS Word comment in which you can type your remarks.
An MS Word comment is typically displayed as a red balloon in the right margin of the document and usually starts with the abbreviation of your name and the timestamp at which the comment is being written. Depending on your version of MS Word, use the following steps for inserting a comment:

MS Word 2007 and MS Word 2010:

1. Select the text you would like to comment upon
2. Open the **Review** ribbon, select **New Comment** in the **Comments** section
3. In the balloon that appears in the right margin, type your comment
4. Click anywhere in the document to continue editing the document

MS Word 2003:

1. Select the text you would like to comment upon
2. From the **Insert** menu, select **Comment** (or click on the **New Comment** button on the **Reviewing** toolbar)
3. In the balloon that appears in the right margin, type your comment
4. Click anywhere in the document to continue editing the document

The text will have coloured lines surrounding it, and a dotted coloured line will connect it to the comment. To delete a comment, simply right click on the balloon and select **Delete Comment**.

- Please do not use the MS Word "track changes" tool and do not write your comments as plain text in the MS Word file.
- In case that you want to provide general comments or remarks that are not specific to a part of the text of this document, please provide them into a separate document and/or e-mail.

In case that you need to translate this document to another language, and then translate back your comments to English, please make sure that your comments are provided in the form described above and that they have not been altered or moved to another section of the text during the translation process.

Introduction

This document establishes the common principles, methods, techniques, standards and protocols to be used for realising the *ECRIS Detailed Technical Specifications* and for implementing the ECRIS software.

These have been conceived based on the preliminary analyses that were carried out during the "Inception Phase" of the project, keeping in mind not only the ECRIS legal basis but also the replies received from the Member States to the "Inception Phase Questionnaire" and to the "Technical Architecture Proposals" documents. In addition, the concrete technical proposals were further discussed in an Expert Sub Group Meeting on 22 September 2010 with technical experts of several Member States so as to reach a sufficient level of maturity and cohesion.

The technical principles, standards and protocols described in this document are to be considered final and binding for all software implementations of the *ECRIS Technical Specifications* – implementations produced by the Member States and by the *ECRIS Reference Implementation* to be provided by the European Commission.

Please note that, where necessary, practical examples are provided so as to illustrate the particulars of a given approach. Deliberately, these examples are not necessarily using known ECRIS data elements so as to keep the focus on the technical subject being discussed rather than on the particulars of ECRIS. It is indeed the aim to present technical solutions that are sufficiently general and flexible so as to be able to accommodate current but also future needs of ECRIS.

Decentralised Architecture

Given its nature, ECRIS falls under the category of *Enterprise Integration* projects. In this category different architectural approaches exist, each with its own benefits and shortcomings. On a high level, the approaches can be grouped in the following two:

- Centralised communication (using patterns and technologies such as a central *ESB*)
- Decentralised communication (*Peer-to-Peer* and similar approaches)

In accordance with the ECRIS legal basis and with the answers from the Member States to the "Inception Phase Questionnaire", only the **decentralised communication approach** is being considered.

Even in a decentralised communication approach, it is possible to propose solutions that still partly use centralised technical artefacts such as common tools (for example naming directories, sequencers, validation tools, etc.) and specific mandatory or optional functions (such as logging, monitoring, collection of statistical data, transliteration, translation, time synchronisation etc.)

In accordance with the ECRIS legal basis and with the answers from the Member States to the "Inception Phase Questionnaire", only a **fully decentralised** communication approach is considered for implementing ECRIS. No centralised technical artefacts or functions are to be used.

Please note that in particular, as a consequence of the fully decentralised approach, time synchronisation of the clocks of the computer servers (for example using NTP servers) falls under the responsibility of each individual Member State. Moreover, it has to be noted that time mismatches between the ECRIS servers can potentially affect the collection of statistics of the message exchanges and create inconsistencies. In order to mitigate that risk, the descriptions of the statistics procedures needs to foresee a percentage of tolerance for such borderline errors. This is out of scope of this specific document and will be dealt with in the "Logging, Monitoring and Statistics Analysis" document.

Fully Decentralised Communication Architecture

Based on the replies provided to the "Inception Phase Questionnaire", the Member States being clearly against using a centralised approach, full or partial, invoked mainly the following aspects to be avoided:

- Increased dependency to external structures
- Increased complexity
- Security concerns
- Creation of a single point of failure

For the sake of completeness, it is to be noted that the centralised model is only endorsed by a few Member States, raising the following benefits:

- Easier troubleshooting in regards to connectivity issues
- Common facilities and services that will be available for all Member States, thus reducing the cost of implementation
- Easier maintenance of common information artefacts
- Easier collection of statistics

Overall, the approach is to keep the same connectivity model as in NJR. Concretely, this means that sTESTA is used as the common communication medium between Member States, leaving the particulars of how connectivity is achieved with sTESTA to each Member State to handle. Each Member States' ECRIS application dialogues with the ECRIS applications of all other Member States, without intermediate software.

The following diagram provides an overview on how connectivity is achieved between Member States using sTESTA. Please note that the local network topology and set-up of each Member State is abstracted and represented as a single entity, further referred to as "Local Member State Network", although in practice it is actually constituted of a chain of specific national and/or local networks.

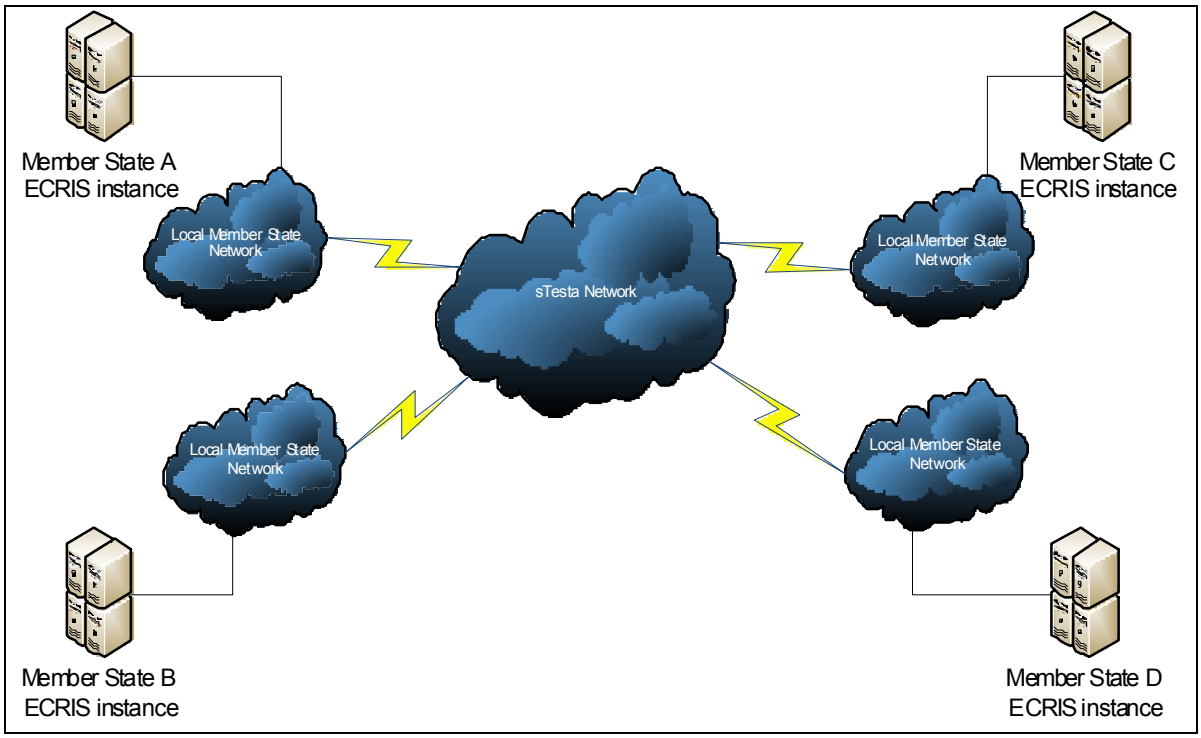


Figure 1 – ECRIS connectivity using sTESTA

Protocols and Standards

It is of utmost importance to establish the required technical protocols and standards that must be respected **completely** by all ECRIS applications, both the national implementations as well as the *ECRIS Reference Implementation*, in order to ensure the technical interoperability of the software systems.

As a result of the answers to the "Inception Phase Questionnaire", all Member States unanimously indicated *Web Services* using *SOAP* as being the preferred *RPC* protocol.

For the sake of completeness, it is interesting to indicate that other standards have been proposed in the questionnaire but were ruled out: *REST*, *XML-RPC*, *CORBA*, *JSON-RPC* and *Etch*.

Web Services using *SOAP* Messages

A *Web Service* is divided in two parts:

1. The *Service Contract* which provides the definition of all the elements required for a message exchange to take place (such as messages, data types, functions, bindings).
2. The actual implementation that supports the functionality described in the *Service Contract*.

Both parts have to comply with the appropriate standards as they are defined by W3C, so that interoperability is guaranteed. Within the ECRIS context and based on the replies received from the Member States to the "Inception Phase Questionnaire" document, the following versions of the protocols and standards are to be used for implementing ECRIS:

- *Xml Schema Definition* (*XSD*) 1.0 in accordance to the following
  - *XML* Schema Part 0: Primer Second Edition (W3C Recommendation 28 Oct 2004)
    http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/

    This version is the latest final version at the time of authoring this document.
  - *XML* Schema Part 1: Structures Second Edition (W3C Recommendation 28 Oct 2004)
    http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

    This version is the latest final version at the time of authoring this document.
  - *XML* Schema Part 2: Datatypes Second Edition (W3C Recommendation 28 Oct 2004)
    http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

    This version is the latest final version at the time of authoring this document.
- Unicode Standard 5.0.2 (to support UTF-8)
  http://www.unicode.org/versions/Unicode5.2.0/
- eXtensible Markup Language (*XML*) 1.0 – Fifth Edition (W3C Recommendation 26 Nov 2008)
  http://www.w3.org/TR/2008/REC-xml-20081126/

  This version is the latest final version at the time of authoring this document.
- Namespaces in *XML* 1.0 - Third Edition (W3C Recommendation 26 Nov 2008)
  http://www.w3.org/TR/2009/REC-xml-names-20091208/

  This version is the latest final version at the time of authoring this document.
- Web Services Description Language (*WSDL*) 1.1 (W3C Note 15 Mar 2001)
  http://www.w3.org/TR/2001/NOTE-wsdl-20010315

  This version is the latest final version at the time of authoring this document.
- Simple Object Access Protocol (SOAP) 1.2, more specifically:
  - SOAP version 1.2 Part 0: Primer – Second Edition (W3C Recommendation 27 Apr 2007)
    http://www.w3.org/TR/2007/REC-soap12-part0-20070427/

    This version is the latest final version at the time of authoring this document.

– SOAP version 1.2 Part 1: Messaging Framework – Second Edition (W3C Recommendation 27 Apr 2007)
http://www.w3.org/TR/2007/REC-soap12-part1-20070427/

This version is the latest final version at the time of authoring this document.

– SOAP version 1.2 Part 2: Adjuncts – Second Edition (W3C Recommendation 27 Apr 2007)
http://www.w3.org/TR/2007/REC-soap12-part2-20070427/

This version is the latest final version at the time of authoring this document.

The aforementioned list of standards is considered as the current mainstream standard in *Web Services*. These are fully supported by main industry-level standard tools and implementations. Additionally, these specifications are currently considered as mature since they have reached final state between 2004 and 2008.

Please note that the aforementioned protocols and standards form the smallest common compatible denominator between the different technical platforms supported by the Member States, with the exception of the PHP platforms which do not fully support some or parts of the standards listed (at least not out-of-the-box nor with any of the open source non-proprietary tools that are available). This implies that adoption of these specifications requires a change or upgrade of that platform, at least partially, so as to ensure proper technical interoperability.

The .NET framework (at least from version 2 and upwards) fully supports the specifications provided.

As for the Java platform, the following matrix shows that all major implementations are fully compatible with these standards (matrix taken from http://wiki.apache.org, the complete matrix can be found under http://wiki.apache.org/ws/StackComparison):

Standard	Axis 1.x	Axis 2.x	CXF	IBM WAS 7.x	JBossWS	XFire 1.2	Metro	Oracle AS 10g
SOAP 1.1	✔	✔	✔	✔	✔	✔	✔	✔
SOAP 1.2	✔	✔	✔	✔	✔	✔	✔	✔
WSDL 1.0	✔	✔	✔	✔	✔	✔	✔	✔
WSDL 1.1	✔	✔	✔	✔	✔	✔	✔	✔

Please note that even though *Web Services* were designed to ensure interoperability between heterogeneous technical environments, quite often this is not the case as different tools do not interpret or implement in the same way the protocols and standards provided. This fact can affect the ECRIS *Detailed Technical Specifications* in two ways:

1. During the design phase, although the *service contract* and its artefacts are deemed valid by the standards of a given design tool, such as for example "Altova XMLSpy", it can be the case that this same *service contract* is not considered valid by other tools implementing the same technical standards (such as for example tools used for generating programming code). Indeed, some tools are less strict in their implementation of the standards than others. For example "XMLSpy" and the ".NET "svcutil" tools allow multiple namespace imports using the <wsdl:import> tag whereas most Java tools consider this as an error. This can block the developers in using the *service contract* for generating the code of their *web service* implementation.

   In order to avoid this specific issue, the *service contract* and all related artefacts (i.e. mainly the XSD files) need to be thoroughly validated and tested using the different platforms used by the Member States, mainly Java/J2EE and the .NET framework.

2. During runtime, interoperability issues can lead to corrupted information or even total loss of service between two communicating entities.

   In order to mitigate this issue, thorough positive and negative tests need to be carried out on the software implementing the *service contract*. Although the definition of the test cases is out of scope of this document, it is worth mentioning that a particular focus should be put on verifying:

   - The validity of the structure of the SOAP messages in regards to SOAP 1.2
   - The ability of an implementation to properly consume and produce "SOAPFault" exceptions, either custom or generic as defined in the *service contract*.
   - The *XML* structure of the *payload* of the messages that an implementation produces and consumes. More specifically the tests should examine:
     - Appropriate namespace declaration in generated messages, including single and multiple declarations of namespaces as attributes in a single element. The implementations do not necessarily need to support both ways of declaration since both are equally valid according to the technical standard.
     - Appropriate namespace scoping in generated messages.
     - Whether the implementation can consume generated messages with all possible ways of declaring namespaces.
     - Whether the implementation can properly generate and validate messages created from XSDs which contain the tag <xsd:choice>
   - The handling of data types, with focus on dates, numeric and complex custom types (such as types created using the <xsd:union> tag or types substituted using the "substitutionGroup" attribute) and enumerated lists of values.

Communication Mode

The communication mode to be used for ECRIS follows closely what is done in the NJR pilot project: "synchronous" technical calls for delivering the *XML* messages and "asynchronous" functional responses.

Synchronous Technical Calls and Asynchronous Functional Responses

The communication mode to be used for the ECRIS implementation is the following:

- The technical transmission of each message is performed by a *web service* which in nature is a **synchronous** remote call. This means that the transmission process of the sending system is blocked and waiting until the call returns a value that informs of correct or incorrect completion of the call on the receiver's side. The message being sent is an *XML* document. The **validation of the *XML* message against the XSD definitions is to be done synchronously** on the call of the *web service*.

- Since the processing of each message cannot be performed immediately but may require human interventions, the **functional** response is provided **asynchronously**. This means that, in response to a synchronous transmission issued by a sender, another synchronous transmission is issued by the receiver later in time in order to carry the functional response to the first transmission. The sending system is not blocked in its execution while waiting for the response to be transmitted. This implies in particular that the messages must carry sufficient information so that they can be related one to another (i.e. message correlation information).

The ECRIS communication mode proposed actually respects the following *Enterprise Integration Patterns*:

- *Point-To-Point channel*, used for the communication between sending and receiving Member States
- *Message* (and various extensions like *Document Message*, *Event Message* and *Command Message*)
- *Message Endpoint*

The following diagram depicts the usage of the patterns mentioned above, in a sample exchange between two Member States, were the central authority of one Member State is requesting information about an individual's criminal record to the central authority of another Member State.

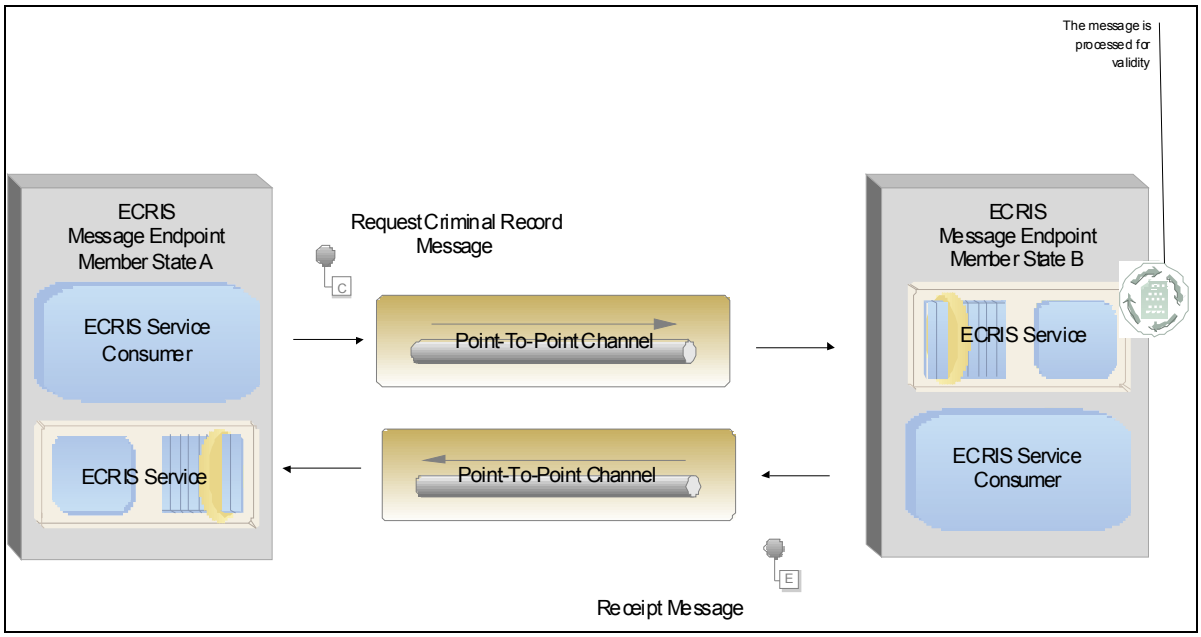


Figure 2 – *Enterprise Integration Patterns* in ECRIS

Error Types and Error Handling

During the message exchanges various types of errors can occur. Based on the nature of these errors, they can be grouped into the following categories:

- Technical errors: these are low-level errors disrupting the transmission of the data between two ECRIS applications and that are returned **synchronously** to the caller of the *web service*. Typical examples are connectivity issues such as request/response time-out, a server being unreachable, unavailability of a technical component, etc. which result in the return of an appropriate HTTP error code. Errors in the validation of the *XML* message against the XSD definition (i.e. errors on structure, cardinality, mandatory values, etc.) are also considered as technical errors and result in the synchronous return of appropriate "SOAP Fault" exceptions. In particular, custom "SOAP Fault" exceptions are to be defined in the *ECRIS Detailed Technical Specifications* with a structure that allows providing sufficient information on the cause of the validation error.

  Such technical errors are typically handled by the IT engineers and/or technical helpdesks.

- Functional Errors: these are the possible results of functional controls that are to be performed on the received data in a second step, after the synchronous *web service* call has returned successfully. Such controls are for example comparison of values against the common reference tables, logical comparison between start and end dates, etc. Errors raised during the functional checks are to be returned in **asynchronous** messages, using appropriate error codes that are to be defined in the *ECRIS Detailed Technical Specifications*.

  Such errors are typically handled by the collaborative work of IT engineers, helpdesks and ECRIS end users.

- Business errors: these errors are exception cases or alternate situations that can arise during the computerised ECRIS dialogues between two Member States' central authorities. These are to be considered as business cases that must be foreseen in the kinematics of messages in the *ECRIS Business Analysis*. Rather than defining errors and error codes, appropriate business messages must be foreseen in the business analysis for dealing with the alternate situations.

This definition of error categories has the advantage of establishing a very clear separation between technical errors and functional errors. More generally, the idea pursued is that all technical errors result in the synchronous return of an HTTP error code or of a specific "SOAP Fault" exceptions while functional errors are to be returned as asynchronous response messages.

Please note that technical server-side processing issues such as bad requests or internal server errors are well defined within the *SOAP* 1.2 specification and are not further elaborated in this chapter. For a complete list of HTTP status codes (i.e. HTTP status codes 2xx, 4xx and 5xx) and their meaning in the context of SOAP 1.2 please visit http://www.w3.org/TR/soap12-part2/#tabreqstatereqtrans.

Please note also that in the NJR pilot project, during a synchronous call between two NJR systems of Member State “A” and “B", the NJR system of the receiving Member State “B” systematically replies with a technical acknowledgment message in order to inform Member State “A” whether the message delivered could be processed correctly from a technical point of view (i.e. ACK/NACK message). This “technical processing” means here that the structure and validity of information contained in the *XML* message sent by “A” is verified by “B” against the rules defined in the *WSDL* file. Later the functional response sent by Member State “B” to Member State “A”, is also performed using a synchronous call which requires an additional technical acknowledgment to be sent back by Member State “A” to “B”. In ECRIS such technical “acknowledgment” messages are not used so as to reduce the complexity of the kinematics. Indeed, the *SOAP* protocol guarantees that, if a synchronous call returns with a value indicating correct execution of the call, the call indeed has been processed correctly by the *web service endpoint* of the receiving software system from a technical point of view. As already defined earlier, the correct processing includes in particular also the validation of the *XML* message against the rules defined in the XSD definitions. In case of incorrect technical processing of a call, the protocol foresees the usage of “SOAP Fault” exception elements that are to be used in ECRIS. For this end, the *ECRIS Detailed Technical Specifications* will define specific and customised “SOAP Fault” exceptions carrying sufficient information related to the cause of the error in the processing, including causes coming from the non-compliance of the *XML* document with the predefined XSD definitions. It has to be noted here that, following best practices of system integration and *SOA* architecture, implementations should not expose internal errors (such as for example failure to connect to an internal database or an internal dependent system) to their callers. Thus, implementations should either respond with HTTP status code 500, which in the *SOAP* 1.2 is used to signify an internal server error, or use a custom “SOAP Fault” exception for notifying the caller that the service is unavailable. Both possibilities are supported in the ECRIS architecture and should be understood by callers, leaving thus to the implementer the decision of the most appropriate solution for his technical platform.

Let's also further illustrate the different handling of functional and business errors. Let's assume that Member State "A" sends a notification message to Member State "B" and that this message contains a reference to a sanction code defined in the common reference tables. Let's now assume that the code provided for the said sanction is incorrect. Member State "B" will only detect the error after the message is properly received during the functional validation phase (i.e. the *XML* message is valid from a structural point of view). At this point, Member State "B" replies to Member State "A" with a functional error message so as to indicate the wrong value that was provided for the sanction. The following UML sequence diagram depicts this transaction:

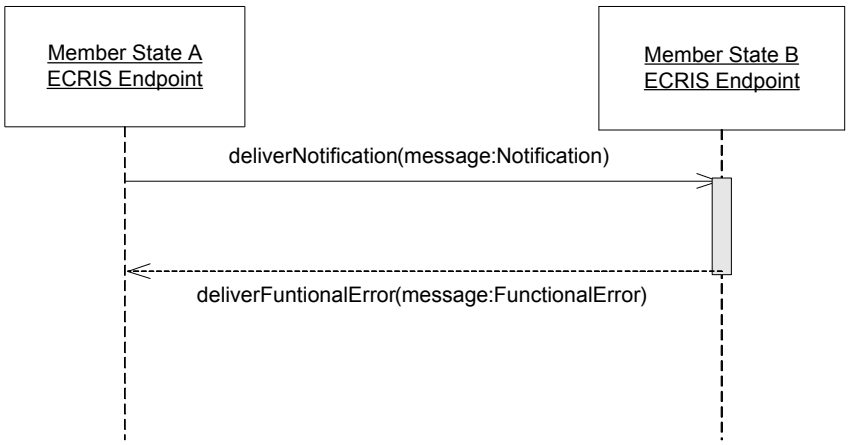


Figure 3 – An unsuccessful transaction due to a functional error

Let's reuse the previous example and assume now that the message was received properly (i.e. technically correct in terms of structure) and that it also contains valid values and passes successfully the functional controls. Let's assume however that the notification refers to a person

that has deceased and that the conviction can thus not be registered in the criminal records. A business message is delivered to Member State "A" so as to notify of the situation. The following UML sequence diagram depicts this transaction:

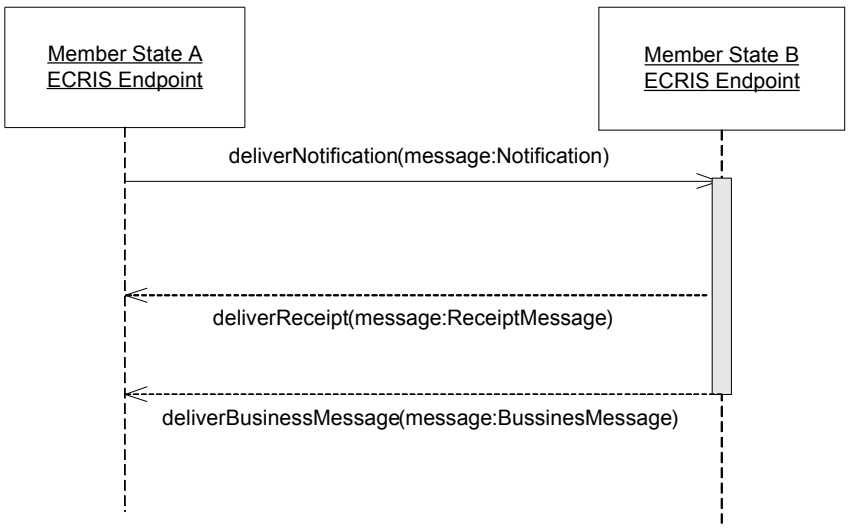


Figure 4 – A foreseen business message is delivered

Message Validity

As a standard good practice, each message's *XML payload* has to be validated against the XSD schema definitions **before being transmitted** in the communication channel by the sending application. This will ensure a higher level of acceptance of the messages received in regards to their structural validity (this does not automatically imply that messages will also be valid in regards to functional validation rules though) as well as reduce the unnecessary message chattering using machine resources such as network bandwidth, serialisation and de-serialisation processing and message validation on the receiver's part.

Connectivity

Connectivity issues have been identified as being quite tricky to handle in the NJR pilot project. Given that ECRIS and NJR are software applications, the capability to react regarding connectivity problems is minimal since they cannot intervene on the *OSI Layer 3* problems. Furthermore, both systems use the same decentralised architecture, using multiple different networks through which the messages are channelled.

However, in order to be able to detect losses of connectivity between two peers, monitoring services can be foreseen.

In the first version of the ECRIS technical specifications, these monitoring services are limited to a simple "isAlive" *web service* to be defined in the *ECRIS Detailed Technical Specifications* in order to keep the implementations simple, given the short timeframe that is available until April 2012. The "isAlive" service is to be implemented in the same web service end-point as the other ECRIS web services and indicates to the caller if the end-point is functional and able to respond to calls. Using this "isAlive" service before transmitting a message so as to confirm the target host's status is optional but recommended. It is also recommended that in the case of multiple message exchanges are to be issued within a short time period, this function should only be used once and not systematically before each *web service* call so as to avoid unnecessary additional traffic in the network and unnecessary consumption of server and network resources on both communicating parties.

Please note that peer services providing additional functionality such as elaborate monitoring of the status of the various *web services*, monitoring of the connectivity between various Member States' ECRIS applications, automated sending of administrative messages, etc. should be kept aside for later versions of the ECRIS technical specifications. Such services are not considered as essential for starting the ECRIS data exchanges in April 2012. At first, organisational measures and precautions should be taken, such as setting up a central entity communicating on planned down-times of servers, helpdesks for trouble-shooting sTESTA issues, etc. The definition of such organisational measures is not in the scope of this document.

***WSDL*/*XSD*/*XML* design principles**

In addition to the definition of the communication architecture and appropriate versions of protocols and standards to be used, also defining design principles in regards to the *service contract* artefacts is essential so as to achieve proper interoperability between software implementations and ensure that the quality of the final products will be sufficient.

The following sections describe the design principles to be followed for producing the *service contact* and *XML* schema definitions of ECRIS.

XSD/XML *naming convention based on ISO 11179*

One of the most important tasks when creating *XSDs* (on which all *XML* documents are then in turn based) is to use naming conventions which will eventually provide well formatted documents that are easy to read and easy to understand, also for human actors. This is essential for ensuring **semantic** interoperability.

Especially within the ECRIS context, defining proper rules for naming conventions is of importance since the technical specifications are expected to further evolve in time and also because these specifications will need to be implemented by many and various persons with different cultures, backgrounds and knowledge.

The *ISO/IEC 11179* – and in particular "ISO/IEC 11179-4: Formulation of data" and "ISO/IEC 11179-5: Naming and identification principles" – definition provides a solid basis on how to build naming conventions. While saying this, please note that it is not the intention here to establish *ECRIS Detailed Technical Specifications* that will necessarily be fully compliant with this ISO standard. It is rather the idea to use this standard as a guideline in order to produce the required technical artefacts.

Based on the aforementioned ISO standard, the following requirements and recommendations in regards to definitions of *XSD*/*XML* types and elements are established:

- A definition shall:
  A. be written in British English
  B. be stated in the singular form (unless the name itself is a plural, for example "premises")
  C. state what the concept **is** rather than what it is not
  D. be stated as a descriptive phrase or sentence(s)
  E. not contain connecting words such as "of", "or", "and", "the", etc.
  F. contain only commonly understood abbreviations
  G. be expressed without embedding definitions of other data or underlying concepts
  H. contain only alphanumeric characters
- Alongside with the aforementioned requirements, a definition should:
  A. state the essential meaning of the concept
  B. be precise and unambiguous
  C. be concise
  D. be able to stand alone
  E. be expressed without embedding rationale, functional usage, or procedural information
  F. avoid circular reasoning
  G. use the same terminology and consistent logical structure for related definitions
  H. be appropriate for the type of metadata item being defined

Using the above recommendations and requirements for *XSD*/*XML* definitions, the

"**Object:Property**" scheme is to be used for naming elements, where:

- **Object** is the class term (for example "person", "authority", "country", etc.)
- **Property** is an element belonging to the referenced class term (for example "first name", "surname", "postal code", etc.)

The "CamelCase" format is to be applied: all words are joined without using spaces or special characters for separating them, with each element's initial letter capitalised within the compound and the first letter is either upper or lower case — as in "LaBelle" or "BlackColor".-As a general rule, upper initial letters are to be used for *XML* entities such as "PersonName", "Offence", "DecisionDate", etc.) and lower case initial letters for *XML* attributes such as "id", "languageCode", etc.

Example: Designing a "Person" class

Let's assume that a complex data type needs to be created so as to represent a "person" type. To that complex type, the properties "name" and "surname" are recognised. For the sake of simplicity, let's assume that "name" and "surname" are simple text representations of a person's name and surname. Following the conventions defined above, the appropriate name for the type to be defined is "PersonType": indeed the class is "Person" and its representation is "Type". Furthermore, this definition contains two properties, "PersonName" and "PersonSurName". This definition, in XSD, would look like the following:

```
<xs:complexType name="PersonType">
    <xs:sequence>
        <xs:element name="PersonName" type="xs:string" />
        <xs:element name="PersonSurName" type="xs:string" />
    </xs:sequence>
</xs:complexType>
```

Example 1 – A "PersonType" definition

Now that the appropriate type definition is created, let's also define an instance of a "person" type that can be used in any *XML* document. The final XSD schema would then look like the following:

```
<xs:complexType name="PersonType">
    <xs:sequence>
        <xs:element name="PersonName" type="xs:string" />
        <xs:element name="PersonSurName" type="xs:string" />
    </xs:sequence>
</xs:complexType>
<xs:element name="Person" type="PersonType" />
```

Example 2 – Defining an element based on "PersonType"

Following that approach, an *XML* document that would include some instances of the element "person", would look like the following:

```
<People>
        <Person>
                <PersonName>John</PersonName>
                <PersonSurName>Smith</PersonSurName>
        </Person>
        .
        .
        .
</People>
```

Example 3 – *XML* produced using "Person" element

Using Object-Oriented Paradigms in XML

XML schemas offer a powerful set of tools for constraining and formalising the vocabulary and grammar of *XML* documents. It is obvious that the structure of an *XML* document, as defined by their schemas, must be created and stored in an organised manner. Developers experienced in object-oriented design know that a flexible architecture ensures consistency throughout the system and helps to accommodate extensibility, flexibility, and modularity.

In order to benefit from such advantages, the following object-oriented paradigms will be applied when designing the *XML* schemas.

Please note that the aim of using such paradigms is to help rationalising the *XML* documents without actually rendering their processing more complicated. Such paradigms usually help reducing development time and logical errors.

Encapsulation

The term “encapsulation” refers in the context of *XSD* and *XML* to the definition of predefined data types where each element bundles and carries its own properties as internal elements. Such data elements should be accessible within the *service contract* simply by referencing their type.

Using the example from the previous chapter, the "PersonType" element defines that a “person” has two properties "PersonNameText" and "PersonSurnameText". The "PersonType" element thus encapsulates all the information relevant to a “person”.

A good practice is to place generic data types such as "PersonType" in a generic schema, usually referred to as “data dictionary”, which contains all the generic data types to be re-used in different schemas. Since such a generic schema is actually only a library of generic data types, no root element is usually required. Instead, it just contains a collection of complex elements defining the structure and content of each generic component.

In order to be reference-able, such an element must be a “global” element meaning that its declaration is an immediate child of the <schema> tag in the *XSD*. Please note that a “local” element refers to an element declaration that is nested within another component; for example, "PersonName" is a “local” element within the “global” element "PersonType".

The main advantage of this practice is the ease of reuse of common structures and the fact that changes to those structures only happen in one place. Another major advantage is that components can be added to this data types’ library as the business expands. Using encapsulation allows for good and logical organisation of the data types definitions, flexibility and standardisation.

Even though two technical ways are provided in the standard W3C *XML* Schema specification for accessing the data types’ library in a schema, only the <xs:import> tag will be used in ECRIS instead of <xs:include>. The reason is that <xs:import> is considered the most appropriate approach for complex designs because included data types preserve their original namespace and thus are clearly separated from local types. Another, even more important, reason for using <xs:import> instead of <xs:include> is the fact that the implementation of the versioning to be used in ECRIS (and defined later in this document) relies on using namespaces in order to denote different versions of *XML* schemas. Thus, if the namespace declaration is lost, the versioning implementation becomes impossible.

Example: Using encapsulation and <xs:import>

In extension to the example provided in chapter 3.4.1 in which a "PersonType" element was defined, let's assume now that the schema containing "PersonType" has target namespace "http://example.com/commons". Furthermore, let's assume that a dictionary needs to be added, represented by the type "DictionaryType". Since this type represents a specific business requirement, it is decided that is not suitable to include it in the same namespace as "PersonType". The resulting *XSD* would then look like the following:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                elementFormDefault="qualified"
                attributeFormDefault="unqualified"
                xmlns:commons="http://example.com/commons">
  <xs:import namespace="http://example.com/commons" schemaLocation="commons.xsd" />
  <xs:complexType name="DictionaryType">
     <xs:sequence>
         <xs:element name="DictionaryName" type="xs:string" />
         <xs:element name="Person" type="commons:PersonType" />
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```

Example 4 – Encapsulation with <xs:import>

Inheritance

Reuse is another useful paradigm of object-oriented design. In software development reuse is achieved through inheritance which, in programming languages is provided by the usage of "subclasses". In *XML* schemas this can be achieved by specifying derivations of existing data types using <xs:extension>.

Such derivations of data types can be realised either by "extension", meaning by adding new properties and/or attributes, or by "restriction", meaning by leaving out specific properties and/or attributes.

Example: Inheritance through derivation by extension

Let's assume that it is required to define a "ProgrammerType" element for representing a programmer. Obviously, a programmer is a person and thus has a name and a surname. In addition, a programmer also knows some programming languages. The new “ProgrammerType” element is defined as an extension of the “PersonType” element, to which a new property “ProgrammerProgrammingLanguages” is added.

The newly created schema would look like the following:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:commons="http://example.com/commons"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified"
           targetNamespace="http://example.com/commons">
    <xs:complexType name="PersonType">
        <xs:sequence>
            <xs:element name="PersonName" type="xs:string" />
            <xs:element name="PersonSurName" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ProgrammerType">
        <xs:complexContent>
            <xs:extension base="commons:PersonType">
                <xs:sequence>
                    <xs:element name="ProgrammerProgrammingLanguages" type="xs:string" />
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:schema>
```

Example 5 – Inheritance with <xs:extension>

Example: Inheritance through derivation by restriction

Let’s assume now that the "PersonType" has been defined as a person having multiple addresses and multiple phone numbers. For that purpose, the properties "PersonAddress" and "PersonPhoneNumber" have been added. The "PersonType" has been modified into something like the following:

```
<xs:complexType name="PersonType">
  <xs:sequence>
    <xs:element name="PersonName" type="xs:string" />
    <xs:element name="PersonSurName" type="xs:string" />
    <xs:element name="PersonAddress" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="PersonPhoneNumber" type="xs:int" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
```

Example 6 – "PersonType" definition

Let's assume now that it is required to define a type for representing students. Each student is a person and thus has a name and a surname but the business rules define that a student can have only one address and one phone number.

The "StudentType" can be derived from the "PersonType" element by restriction and would look like the following:

```
<xs:complexType name="StudentType">
    <xs:complexContent>
        <xs:restriction base="commons:PersonType">
            <xs:sequence>
                <xs:element name="PersonName" type="xs:string" />
                <xs:element name="PersonSurName" type="xs:string" />
                <xs:element name="PersonAddress" type="xs:string" />
                <xs:element name="PersonPhoneNumber" type="xs:int" />
            </xs:sequence>
        </xs:restriction>
    </xs:complexContent>
</xs:complexType>
```

Example 7 – Inheritance using <xs:restriction>

Polymorphism

"Polymorphism" means the ability to assign a different meaning or usage to something in different contexts – specifically, to allow an object to have more than one form.

In object-oriented programming languages polymorphism implies a different reaction to an input. More specifically, it is the ability of a subclass to respond differently to the same input. Since *XML* is not a behavioural language, polymorphism occurs at the level of properties and attributes.

Example: Polymorphism in *XML*

Based on the previous example of derivation by restriction, let's assume that the business model prohibits a student's address to have a length greater than 20 characters. This can easily be realised by creating a new simple type called "LimitedStringType" which is derived from the base "xs:string" type and then use this new type instead of the base in the definition of the property "PersonAddress".

The resulting schema would then look like the following:

```
<xs:complexType name="StudentType">
   <xs:complexContent>
      <xs:restriction base="commons:PersonType">
         <xs:sequence>
            <xs:element name="PersonName" type="xs:string" />
            <xs:element name="PersonSurName" type="xs:string" />
            <xs:element name="PersonAddress" type="commons:LimitedStringType" />
            <xs:element name="PersonPhoneNumber" type="xs:int" />
          </xs:sequence>
      </xs:restriction>
   </xs:complexContent>
</xs:complexType>

<xs:simpleType name="LimitedStringType">
     <xs:restriction base="xs:string">
          <xs:maxLength value="20" />
      </xs:restriction>
</xs:simpleType>
```

Example 8 – Polymorphism with type substitution

Abstraction using "xsi:type"

Another aspect of polymorphism is "abstraction", which is the ability of a type "A" to appear and be used like another type "B".

In object-oriented programming, this allows to manipulate successfully an object instance as long as its base object type is known, without necessarily caring about the specifics of the object itself. In practice this is achieved by using inheritance and programming the logic using the parent classes rather than the subclasses when possible.

A major advantage of abstraction is that it allows defining several different extensions to a class but still reuse the same programming logic for implementing the common behaviour. It makes it easier in particular to reach compatibility throughout different versions of the system.

This section proposes to implement abstraction in the ECRIS *XML* schemas/*XML* documents by combining the previous design principles and using the "xsi:type" attribute (which is part of the http://www.w3.org/2001/XMLSchema-instance namespace).

Indeed, this combination of design principles, usage of "xsi:type" and namespaces allows including elements of the same type **but of different versions** in the same structure without rendering this structure invalid. The main benefit is that it avoids the need to transform information previously received into a newer format since previous versions of the element structure remain compatible.

Another advantage is that it facilitates the processing, for example when using XPATH or derivatives of it.

Please note that the use of the "xsi:type" attribute in conjunction with parsing libraries used to serialise or de-serialise *XML* documents into in-memory objects can be problematic, depending on how appropriate and complete the parsing software actually implements the W3C standards. Most commonly this is the case for older versions of such parsers, since software vendors fix such issues as soon as they are recognised. For example, Apache Tuscany (http://tuscany.apache.org/) had a well-known issue handling "xsi:type", which was resolved in a fix in 2007.

It is thus recommended that implementers of the ECRIS technical specifications should check their *XML* parsing software to verify that the implementation is appropriate and if not update this software to a later version which fixes such issues.

Example: Using abstraction allows easier navigation

Let's assume that a type named "AssociationType" has been defined for representing relations between elements. This type has the properties "AssociationSourceID" and "AssociationDestinationID" which point to the identifiers (aka ID's) of referenced elements. For the sake of simplicity, let's not consider at this point the type of those ID's. By its definition, this relation is N-to-N, from multiple sources to multiple destinations. Let's assume now that it is necessary to define a type for N-to-1 relations (i.e. multiple sources to one destination or *Many-to-One*) and a type for 1-to-N relations (i.e. one source to multiple destinations or *One-to-Many*). To that end, as described previously in this document, it is possible to define two types derived from the "AssociationType" by restriction called "ManyToOneAssociationType" and "OneToManyAssociationType".

The XSD then would look like the following:

```
<xs:complexType name="AssociationType">
        <xs:sequence>
                <xs:element name="AssociationSourceID" type="xs:string" minOccurs="1" maxOccurs="unbounded" />
                <xs:element name="AssociationDestinationID" type="xs:string" minOccurs="1" maxOccurs="unbounded" />
        </xs:sequence>
</xs:complexType>
<xs:complexType name="ManyToOneAssociationType">
        <xs:complexContent>
                <xs:restriction base="AssociationType">
                        <xs:sequence>
                                <xs:element name="AssociationSourceID" type="xs:string" minOccurs="1" maxOccurs="unbounded" />
                                <xs:element name="AssociationDestinationID" type="xs:string" minOccurs="1" maxOccurs="1" />
                        </xs:sequence>
                </xs:restriction>
        </xs:complexContent>
</xs:complexType>
<xs:complexType name="OneToManyAssociationType">
        <xs:complexContent>
                <xs:restriction base="AssociationType">
                        <xs:sequence>
                                <xs:element name="AssociationSourceID" type="xs:string" minOccurs="1" maxOccurs="1" />
                                <xs:element name="AssociationDestinationID" type="xs:string" minOccurs="1" maxOccurs="unbounded" />
                        </xs:sequence>
                </xs:restriction>
        </xs:complexContent>
</xs:complexType>
```

Example 9 – "AssociationType" and children types definitions

Based on the *XSD* definitions above, the *XML* document containing various relations would look like the following:

```
<Associations>
        <Association>
                <AssociationSourceID>aSourceId</AssociationSourceID>
                <AssociationSourceID>bSourceId</AssociationSourceID>
                <AssociationSourceID>cSourceId</AssociationSourceID>
                <AssociationDestinationID>aDestinationId</AssociationDestinationID>
                <AssociationDestinationID>bDestinationId</AssociationDestinationID>
                <AssociationDestinationID>cDestinationId</AssociationDestinationID>
                <AssociationDestinationID>dDestinationId</AssociationDestinationID>
        </Association>
        <ManyToOneAssociation>
                <AssociationSourceID>aSourceId</AssociationSourceID>
                <AssociationSourceID>bSourceId</AssociationSourceID>
                <AssociationSourceID>cSourceId</AssociationSourceID>
                <AssociationDestinationID>aDestinationId</AssociationDestinationID>
        </ManyToOneAssociation>
        <OneToManyAssociation>
                <AssociationSourceID>cSourceId</AssociationSourceID>
                <AssociationDestinationID>aDestinationId</AssociationDestinationID>
                <AssociationDestinationID>bDestinationId</AssociationDestinationID>
                <AssociationDestinationID>cDestinationId</AssociationDestinationID>
                <AssociationDestinationID>dDestinationId</AssociationDestinationID>
        </OneToManyAssociation>
</Associations>
```

Example 10 – Sample *XML* using "AssociationType" and children definitions

Of course, the resulting *XML* is well formatted and easy to read, but as soon as processing needs to be done using some form of XPATH (or derivatives of XPATH or tools internally using XPATH) for retrieving all associations in the content regardless of their specific type, three different expressions ("//Association", "//ManyToOneAssociation" and "//OneToManyAssociation") are required.

To avoid this, the *XML* document can also be written like the following:

```
<Associations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
        <Association>
                <AssociationSourceID>aSourceId</AssociationSourceID>
                <AssociationSourceID>bSourceId</AssociationSourceID>
                <AssociationSourceID>cSourceId</AssociationSourceID>
                <AssociationDestinationID>aDestinationId</AssociationDestinationID>
                <AssociationDestinationID>bDestinationId</AssociationDestinationID>
                <AssociationDestinationID>cDestinationId</AssociationDestinationID>
                <AssociationDestinationID>dDestinationId</AssociationDestinationID>
        </Association>
        <Association xsi:type="ManyToOneAssociationType ">
                <AssociationSourceID>aSourceId</AssociationSourceID>
                <AssociationSourceID>bSourceId</AssociationSourceID>
                <AssociationSourceID>cSourceId</AssociationSourceID>
                <AssociationDestinationID>aDestinationId</AssociationDestinationID>
        </Association>
        <Association xsi:type="OneToManyAssociationType ">
                <AssociationSourceID>cSourceId</AssociationSourceID>
                <AssociationDestinationID>aDestinationId</AssociationDestinationID>
                <AssociationDestinationID>bDestinationId</AssociationDestinationID>
                <AssociationDestinationID>cDestinationId</AssociationDestinationID>
                <AssociationDestinationID>dDestinationId</AssociationDestinationID>
        </Association>
</Associations>
```

Example 11 – Sample *XML* using "xsi:type" to abstract type definitions

Please note that with the usage of the “xsi:type” attribute, it is now possible to request all available associations from the *XML* document with a single XPATH expression ("//Association" for retrieving all associations recursively). On top of that, the *XML* still validates properly against the *XSD* schema defined earlier.

Example: Using abstraction for inserting new extensions into an existing structure

Let’s consider again the previous example defining a dictionary of persons. As a reminder, the “DictionaryType” element contains a property "PersonType" for the list of persons to be contained in the dictionary.

Let's now assume that it is required to also add programmers into this dictionary. The most straightforward solution is to add a new property in the "DictionaryType" definition. While this is very simple, it however implies necessarily changes into the implementations of the consumers of this dictionary type.

Instead of changing the dictionary type definition, the *XML* documents simply need to be produced slightly differently, like the following:

```
<commons:Dictionary xsi:schemaLocation="http://example.com/commons commons.xsd"
                    xmlns:commons="http://example.com/commons"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance">
    <commons:Person xsi:type="commons:PersonType">
        <commons:PersonName>John</commons:PersonName>
        <commons:PersonSurName>Smith</commons:PersonSurName>
        <commons:PersonAddress>Cypress Road, 12</commons:PersonAddress>
        <commons:PersonPhoneNumber>+44321233111</commons:PersonPhoneNumber>
    </commons:Person>
    <commons:Person xsi:type="commons:ProgrammerType">
        <commons:PersonName>John</commons:PersonName>
        <commons:PersonSurName>Doe</commons:PersonSurName>
        <commons:PersonAddress>Cypress Road, 13</commons:PersonAddress>
        <commons:PersonPhoneNumber>+446545677</commons:PersonPhoneNumber>
        <commons:ProgrammerProgrammingLanguages>Perl, PHP, C++, Scala, Java, .NET</commons:ProgrammerProgrammingLanguages>
    </commons:Person>
</commons:Dictionary>
```

Example 12 – Sample *XML* using "PersonType" and "ProgrammerType"

Please note that this *XML* is perfectly valid against the *XSD* and unless the consumer explicitly needs to use the new property "ProgrammingLanguages" of programmers, no changes are required in the implementation.

Example: Using abstraction for inserting new versions of extensions into an existing structure

Let's assume now that it is necessary to version the type definitions. Without delving yet into the versioning aspects as such (these are elaborated in detail later in this document), let's assume that major and minor version numbers are used in the namespace declaration.

Let's take the following *XSD* definition that is stored in a file named "commons-v1.0.xsd":

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified"
           xmlns:commons="http://example.com/commons-v1.0"
           targetNamespace="http://example.com/commons-v1.0">

    <xs:complexType name="PersonType">
        <xs:sequence>
            <xs:element name="PersonName" type="xs:string" />
            <xs:element name="PersonSurName" type="xs:string" />
            <xs:element name="PersonAddress" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
            <xs:element name="PersonPhoneNumber" type="xs:int" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ProgrammerType">
        <xs:complexContent>
            <xs:extension base="commons:PersonType">
                <xs:sequence>
                    <xs:element name="ProgrammerProgrammingLanguages" type="xs:string" />
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:element name="Dictionary">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Person" type="commons:PersonType" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

Example 13 – "ProgrammerType" definition

Let's assume now that it is required to modify the definition of "ProgrammerType" so that instead of using a free text field for transmitting the programming languages, a specific collection of values is used. Let's also assume that this change is done in a new version v1.1 of the *XSD* definitions, in a file called "commons-v1.1.xsd":

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:commons-v1.0="http://example.com/commons-v1.0"
           xmlns:commons-v1.1="http://example.com/commons-v1.1"
           targetNamespace="http://example.com/commons-v1.1"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
    <xs:import namespace="http://example.com/commons-v1.0" schemaLocation="commons-v1.0.xsd"/>
    <xs:complexType name="ProgrammerType">
        <xs:complexContent>
            <xs:extension base="commons-v1.0:PersonType">
                <xs:sequence>
                    <xs:element name="ProgrammerProgrammingLanguages" type="commons-v1.1:ProgrammingLanguagesType" />
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    <xs:simpleType name="ProgrammingLanguagesType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="PHP" />
            <xs:enumeration value="JAVA" />
            <xs:enumeration value=".NET" />
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Example 14 – New version of "ProgrammerType" definition

Please note that essentially, the principles of polymorphism and extensions are used here for establishing the new version of the XSD, with the notion of versioning being identified in the namespace.

Now, still using the "xsi:type" attribute in the *XML* documents, parties interested in using the new version in addition to the previous version (for example because there is a need to retransmit information that is available in the previous version) are capable of producing *XML* documents like the following:

```
<commons-v1.0:Dictionary xsi:schemaLocation="http://example.com/commons-v1.1 commons-v1.1.xsd"
                         xmlns:commons-v1.0="http://example.com/commons-v1.0"
                         xmlns:commons-v1.1="http://example.com/commons-v1.1"
                         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <commons-v1.0:Person xsi:type="commons-v1.0:PersonType">
        <commons-v1.0:PersonName>John</commons-v1.0:PersonName>
        <commons-v1.0:PersonSurName>Smith</commons-v1.0:PersonSurName>
        <commons-v1.0:PersonAddress>CypressRoad, 12</commons-v1.0:PersonAddress>
        <commons-v1.0:PersonPhoneNumber>+44321233111</commons-v1.0:PersonPhoneNumber>
    </commons-v1.0:Person>
    <commons-v1.0:Person xsi:type="commons-v1.0:ProgrammerType">
        <commons-v1.0:PersonName>John</commons-v1.0:PersonName>
        <commons-v1.0:PersonSurName>Doe</commons-v1.0:PersonSurName>
        <commons-v1.0:PersonAddress>CypressRoad, 13</commons-v1.0:PersonAddress>
        <commons-v1.0:PersonPhoneNumber>+44655677</commons-v1.0:PersonPhoneNumber>
        <commons-v1.0:ProgrammerProgrammingLanguages>Perl, PHP, C++, Scala, Java, .NET</commons-v1.0:ProgrammerProgrammingLanguages>
    </commons-v1.0:Person>
    <commons-v1.0:Person xsi:type="commons-v1.1:ProgrammerType">
        <commons-v1.0:PersonName>John</commons-v1.0:PersonName>
        <commons-v1.0:PersonSurName>Doe</commons-v1.0:PersonSurName>
        <commons-v1.0:PersonAddress>CypressRoad, 13</commons-v1.0:PersonAddress>
        <commons-v1.0:PersonPhoneNumber>+44655677</commons-v1.0:PersonPhoneNumber>
        <commons-v1.1:ProgrammerProgrammingLanguages>JAVA</commons-v1.1:ProgrammerProgrammingLanguages>
    </commons-v1.0:Person>
</commons-v1.0:Dictionary>
```

Example 15 – Using multiple versions of "ProgrammerType" using "xsi:type"

Messages and XML *document content*

The previous chapters have elaborated on the design principles for **defining the structures** of the *XML* documents to be transmitted between ECRIS applications.

In addition to properly naming, defining and structuring data types and elements, another best practice is to structurally separate functional information that is necessary for the functional processing by the consumer and technical information related to the transmission of the message itself or to specifics of the technical implementation.

"Technical" information is understood here as all data elements that are not directly related to the functional content of the *XML* document – i.e. in the context of ECRIS, that are not directly related to the information on notification of convictions, on requests for information on criminal records data or responses to such requests – but that are used for transmitting necessary information about the transmission process, the transactional behaviour or specifics of the software implementation (for example technical message and/or transaction identifiers, routing information, technical timestamps, etc.).

The information contained in the *XML* messages can be categorised as follows:

- Purely technical information not related to either the dialogue or the functional content of the message such as security tokens, hash codes, etc. Such information is to be placed in a specific *XML* structure named "TechnicalMetaData".
- Technical information that is related to the dialogue between two Member States (for example message ID's, "from-to" information, transaction time-out information, etc.. Such information is to be placed in a specific *XML* structure named "MessageMetaData".
- Functional information, directly related to the business process that the message belongs to (i.e. the business content of the notification, request, response, etc.). Such information is to be placed under a specific *XML* structure named "MessageData".

As an example, let's assume that it has been agreed to use an ID for uniquely identifying any given message being exchanged between two ECRIS applications. In addition, in the case where there is a need to request more information regarding a message previously received, a logical connection needs to be established between the two subsequent messages (i.e. correlation of messages).

In order to do this, two custom properties named "MessageId" of type "MessageIdType" and "InResponseTo" of type "InResponseToType" are defined.

In accordance with categories defined above for separating the technical information from the functional one, these new properties are placed in the separate *XML* construct contained in the message:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:msg="http://ec.europa.eu/ECRIS/messages-v1.0"
                  xmlns:meta="http://ec.europa.eu/ECRIS/messages-metadata-v1.0">
    <soapenv:Head />
    <soapenv:Body>
      <msg:request>
        <meta:MessageMetaData>
          <meta:MessageId>1234</meta:MessageId>
          <meta:TimeStamp>200910150912+0200</meta:TimeStamp>
        </meta:MessageMetaData>
        <msg:MessageData>
          <!-- rest of message ommited for brevity-->
        </msg:MessageData>
    </msg:request>
  </soapenv:Body>
</soapenv:Envelope>
```

Example 16 – Separation of technical information using a separate *XML* structure

The response to this message would look like the following:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:msg="http://ec.europa.eu/ECRIS/messages-v1.0"
                  xmlns:meta="http://ec.europa.eu/ECRIS/messages-metadata-v1.0">
    <soapenv:Head />
    <soapenv:Body>
      <msg:request>
        <meta:MessageMetaData>
          <meta:MessageId>5678</meta:MessageId>
          <meta:InResponceTo>1234</meta:InResponceTo>
          <meta:TimeStamp>201001021345+0200</meta:TimeStamp>
        </meta:MessageMetaData>
        <msg:MessageData>
          <!-- rest of message ommited for brevity-->
        </msg:MessageData>
    </msg:request>
  </soapenv:Body>
</soapenv:Envelope>
```

Example 17 – Reply to the previous message, showing the "InResponseTo" property

Please note that another possibility for structurally separating the technical information from the functional information would have been to use the header part of the SOAP envelope for placing the technical meta-data. However this possibility has been discarded in order to facilitate the implementation and programming work to be performed by the Member States. The SOAP headers are thus not used in the ECRIS technical specifications.

The main drawback of not using SOAP headers is that it requires loading in memory the complete SOAP body for processing the information, also when only technical information needs to be processed at some point. This can have a drawback on performance for large numbers of messages in the case where for example an implementation internally uses an ESB (Enterprise Service Bus) and implements specific routing based only on the message meta-data.
Please note also that existing *web service* extensions such as "WS-Addressing" were considered but discarded since, according to the responses to the "Inception Phase Questionnaire", several Member States do not support such specific *web service* extensions.

Common reference tables

In a way that is similar to what is done in the NJR pilot project, the *ECRIS Detailed Technical Specifications* need to foresee common reference tables in *XML* that define a common codification for predefined lists of possible values for specific *XML* elements. This indeed reduces the efforts required for performing additional transliteration/translation each time an *XML* message is received since the values contained in the common reference tables are typically translated once and then reused throughout all message exchanges.
The reference information that is defined already in such common reference tables must however not be duplicated into the *XML* messages themselves. Only the codes/IDs must be used in the ECRIS *XML* messages. The purpose of the *XML* messages is indeed only to be processed by the ECRIS applications and not to be read by humans.
This approach implies however that the following rule must be defined: in the common reference tables, only additions of records can be allowed. Changes or deletions of information records in a reference table are thus forbidden once such a reference table has been used in the ECRIS data exchanges. For each record, the reference tables must foresee validity periods, using "valid from" and "valid to" dates. Logical removal of a reference value is thus performed by setting the "valid to" date appropriately, indicating that a value is no longer to be used after a set date. A change in a reference value is done by logically removing the old value, setting the "valid to" date as just described, and adding the modified value as a new value in the reference table, setting its "valid from" date appropriately so that it replaces the value that has been marked as invalid.

Documentation

While it is important to properly apply design principles when creating an *XML* Schema or a *service contract*, it is equally important to document accordingly the various entities, properties, types and other significant elements in those two technical artefacts.

Regarding *XML* Schemas, the <xs:documentation> element of the standard <xs:annotation> structure is to be used for providing human-readable documentation. In addition to the fact that using this element allows to provide valuable in-line documentation for future implementers of the specifications, this documentation can also easily be extracted in other formats such as HTML or PDF by using appropriate *XML* style-sheets. The <xs:documentation> elements also allow translating the documentation in different languages, indicated by using the " xml:lang" attribute.

Similarly, for *service contracts* the <wsdl:documentation> element is to be used for adding appropriate documentation. The usage of this element is identical to the usage of the <xs:documentation> element with the exception that it does not support multiple languages. A custom *XML* type will be defined in the *ECRIS Detailed Technical Specifications*, extending the base <wsdl:documentation>, in order to provide support to multiple languages for this element.

Please note that the language codes to be used in the “xml:lang” attribute are the 2-letter codes defined in the ISO 639-1 standard, using lower case. Furthermore, please note that that the documentation provided in the initial version of the *ECRIS Detailed Technical Specifications* by iLICONN will be English.

Versioning

This chapter is divided in two main parts:

- The first subchapter elaborates on the versioning concepts and various possibilities and techniques that can be applied to *XSD*/*XML* and *web services* in order to facilitate the implementation of versioning.
- The second subchapter describes how versioning is implemented on the level of the *ECRIS Technical Specifications*.

Concepts

Any data model usually evolves through time, as the business or functional requirements that this model supports evolve and change. It is rarely the case that a data model defined remains the same throughout its lifecycle. The same is also true for any API or service provided by an application, since functions can change name to represent something new or even change totally their behaviour in order to accommodate the evolution that any software product goes through its lifecycle.
To solve the interoperability issues created from such very much wanted evolutions, proper versioning needs to be considered.

Introduction

In general, versioning provides the concepts, methods and tools so as to handle the interoperability throughout the evolutions of any information entity. More specifically, versioning introduces the following concepts so as to tackle the problems created during such evolutions:

- **Forward Compatibility:** The ability to accept input intended for later versions.
- **Backwards compatibility:** A relationship between two components, where a new component provides all of the functionality of the old component.

These concepts are realised using the following two principles.

- **Extensibility:** A principle where the implementation takes into consideration future growth.
- **Graceful Degradation:** A principle that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components.

The way these concepts are actually implemented, using the principles defined above, is based on the strategy that needs to be defined when versioning is required. This strategy is in itself a design/architecture approach that defines the context within which changes are deemed backwards compatible, whether a system should support forward compatibility, how compatibility is achieved or how degradation is implemented for functionality and/or information that are becoming obsolete.

Please note that the term "degradation", when used in the context of information, usually implies transformation of the information from an old and obsolete format to a new format. With that in mind, it is nearly impossible to elaborate a proper versioning strategy that will allow for no transformation of information, regardless of the changes in structure or content.

Versioning concepts in XML *schema design*

XML document's structure and content definitions as well as validation through the usage of XSD schemas is important in order to ensure that the information exchanged follows the rules that have been agreed by the exchanging parties. Since such rules are susceptible to change in time, version concepts are required so as to be able to create schemas that can properly evolve without creating corruption or major incompatibilities in the *XML* information that has been already exchanged previously or that will be exchanged in the future.

The practices followed in regards to versioning in XSD schemas are design time practices, meaning that they do not require the software using the said schemas for validation to implement a special logic for performing the validation.

These design practices fall into the following high level categories:

- Schema versioning
- Designing for extensibility
- Decoupling dependencies

More specific examples of such practices are presented below.

Schema versioning – using a new *XML* namespace for incompatible version releases

Versioning the "targetNamespace" attribute is a disruptive change, meaning that *XML* document instances will not validate successfully until they are changed to use the new "targetNamespace" value. Since this is a disruptive change it should be used only for major versioning changes in the XSD schema.

Note also that during the de-serialisation process (from *XML* to in-memory instances of software value objects), the code generator of the *XML* parsing software which reads the *XML* document and generates the software value objects uses precisely the namespace for identifying the appropriate classes of the value objects to be created.

As an example, assuming the following definition for the common elements in the ECRIS namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://
ec.europa.eu/ECRIS/common/v1.0">
        .
        .
        .
</xs:schema>
```

Example 18 – Version identifiers in the namespace declaration

Following the principle described above, the next major version would look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://
ec.europa.eu/ECRIS/common/v2.0">
        .
        .
        .
</xs:schema>
```

Example 19 – Changing version identifiers in the namespace declaration

Decoupling dependencies by using proxies

Quite often, it is required to use in *XML* schemas data types that are prone to future changes either in their structure or in their predefined lists of possible values. Direct dependencies of an *XML* schema with such volatile elements result in frequent version iterations, not because the schema itself changed for implementing extra functionality or behaviour, but only for the reason that this specific element, on which the schema depends, changed.

To minimise version iterations, the object-oriented "proxy" design pattern can be applied. The “proxy” is an object acting as an in-between element that allows hiding complexity of the referenced data type, keeping code changes to a minimum when a new version of the encapsulated object is created and defining additional functionality.

Using the principle of polymorphism explained earlier in this document, it is possible to encapsulate a volatile data type by defining an extension. The volatile data type is then imported into the *XML* schema rather than declared in it so that when new versions of the data type are defined, only the import statement is modified.

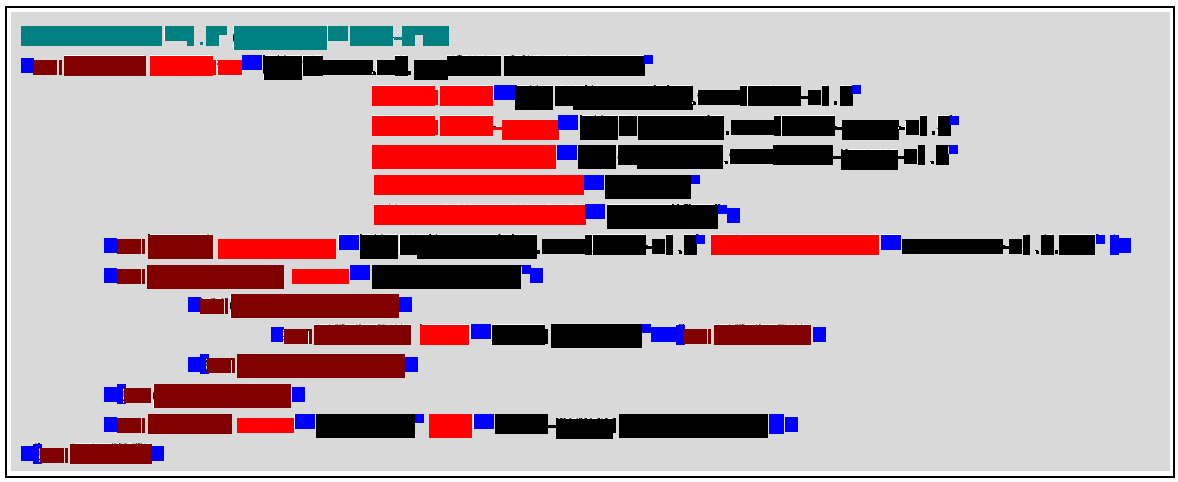
Please note that *XML* documents produced by using the “proxy” technique described here are not completely autonomous, which can have as a result that they may contain at some point invalid references to old data elements.

Let's illustrate this principle by using an example of dependency on an *XML* schema provided by a third party. Let's assume that a list of possible values for cities, on which the *XML* schema relies, is produced and maintained by a third party and looks like the following:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified"
           xmlns:cities="http://cityprovider.com/cities-v1.0"
           targetNamespace="http://cityprovider.com/cities-v1.0">
    <xs:complexType name="CityType">
        <xs:sequence>
            <xs:element name="CityName" />
            <!-- further structure ommited for brevity-->
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="CityNameType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Brussels" />
            <xs:enumeration value="London" />
            <xs:enumeration value="Paris" />
            <xs:enumeration value="Berlin" />
            <!-- further city names ommited for brevity-->
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Example 20 – Schema defining possible values for city names

In order to avoid being dependent on the changes the third-party wishes to implement in his schema, it is possible to define a proxy named "CityProxyType" that extends the "CityType" provided by the third party:



Example 21 – Schema defining proxy type for "CityType"

If at some point in the future the third party decides to add, change or remove default values or even change the structure of the "CityType" element, only an adjustment of the "CityProxyType" will be necessary rather than implementing changes in all other schemas.

Versioning concepts in Web Services

Due to the fact that *web services* are defined in *XML*, they are also candidate for applying the versioning principles and concepts mentioned above. In that respect, different design approaches were progressively established during the years of evolution of *web services*. Such approaches are nowadays common and endorsed by major vendors (such as IBM, Microsoft and Oracle) and do not require any extra software implementation or specific software products to work since they leverage the abilities already provided by the XSD and *WSDL* specifications.
The approaches are briefly presented here so as to give a high level overview of the possibilities.

Using major version number in the *WSDL* target namespace and name of the *WSDL* file

Following this approach, when designing a *web services* contract, the major number of the release is embedded in the *WSDL* target namespace. By encoding only the major release number in the namespace, successive minor releases share the same namespace and so are compatible.
The following snippet illustrates this versioning principle for the "ECRISService" *web service*:

```
<definitions name="ECRISService-v1.0.wsdl"
             targetNamespace="http://ec.europa.eu/ECRIS/service-v1.0"
..
```

Example 22 – Using version numbers for schema file name and target namespace

Versioning data types according to *XSD* conventions and import them into the *WSDL* contract

Generally it is good practice to physically separate the definitions of data-types that a *web service* uses and to place them in separate XSD files, with their own namespaces, rather than bundle them in the types section of a *WSDL* document. In particular, this increases the readability and maintainability of the *WSDL* files and avoids that the *service contract* is cluttered and confusing to the untrained eye.

Based on such practice, whenever the structure of the information transmitted in message exchanges needs to be modified, different approaches can be used for the versioning of the XML schema definitions, such as for example embedding the date of creation, using a major-only or a major/minor scheme (such as for example a –v1 or –v2.1 suffix). While all are suitable for the needs of versioning, each approach has strengths and weaknesses and the choice on which one to use should depend on the principles followed for designing the *service contract*.
The following example gives an overview of using the creation date for versioning the XSD artefacts. Please note in bold the namespace declaration, the schema location for the common types as well as the fact that the *WSDL* contract namespace is using also a major/minor version to reflect these changes.

```
<wsdl:types>
   <xsd:schema targetNamespace="http://ec.europa.eu/ECRIS/service-v1.1"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
         <xsd:import namespace="http://ec.europa.eu/ECRIS/common/2010/08"
                        schemaLocation="./common.xsd"/>
          ...
   </xsd:schema>
</wsdl:types>
```

Example 22 - Using <xsd:import> to import schema definitions in ~~wsdl~~*WSDL*

Encoding major and minor version in the target namespace of the *WSDL* <types> section

According to the practice described above, the data-types used in the *WSDL* interfaces should be versioned separately. The *WSDL* then typically imports these data-types to be used and either defines elements that extend the types provided by the *XML* schema(s) being imported or uses the types directly for defining the *service contract*. When importing the *XML* schema(s), a namespace with an explicit major and minor number should be used.

In the snippet below, please note in bold the change in minor version for the *WSDL* schema:

```
<wsdl:types>
    <xsd:schema targetNamespace="http://ec.europa.eu/ECRIS/service-v1.1"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <xsd:import namespace="http://ec.europa.eu/ECRIS/xsd/common/2010/08"
                    schemaLocation="./common.xsd"/>
        ...
    </xsd:schema>
</wsdl:types>
```

Example 23 – Using version numbers in the target namespace of types defined in the *WSDL* declaration

Using major and minor version in the *web service* interface

In *web services*, the term "interface" refers to the definition of the operations that can be performed and the in- and output messages used. In *WSDL* 1.1 this is also known as "portType" (which has been renamed "interface" in *WSDL* 2.0).

For implementing versioning in *web services,* it is good practice to define an explicit interface for each version of the *web services* and to embed the major and minor version numbers in its the definition.

Successive versions of the "ECRISService" should be named "ECRISService_v1.0", "ECRISService _v1.1", "ECRISService_v1.2" etc. By making the interface version explicit it allows the same code-base to implement different versions of the same *web services* interface.

The following example shows the definition of the "ECRISService" where version 1.0 supports only the “deliverNotification” operation whereas version 1.1 also supports the "requestStatus" operation. Please note also that in this example version 1.1 is backwards compatible, since it supports all the operations provided by the previous version without any change to these operations signatures.

```
<portType name="ECRISService_v1.0">
 <operation name="deliverNotification">
    <input message="tns:notification" name="deliverNotification"/>
    <output message="tns:notificationResponse" name="deliverNotification Response"/>
 </operation>
</portType>
<portType name="ECRISService_v1.1">
 <operation name="deliverNotification">
    <input message="tns:notification" name="deliverNotification"/>
    <output message="tns:notificationResponse" name="deliverNotification Response"/>
 </operation>
 <operation name="requestStatus">
    <output message="tns:statusResponse" name="requestStatusResponse "/>
 </operation>
</portType>
```

Example 24 – Using version numbers in "portType" definition

Versioning “web service endpoints” in accordance to versions of the *web service* interface

The “web service endpoint” (also known as “port” in *WSDL* 1.1) is the declaration in the *WSDL* file of the piece of software that actually implements the *web services* being defined.
Versioning the *web service endpoints* together with the *web services interface* allows different versions of the same interface to be bound to and handled by either one web *service endpoint* or different versions of the same *web service endpoint*.

In the following snippet, the same *web service endpoint* is handling two different versions of the interface "ECRISService". Please note in bold the different versions of the interface actually pointing to the same *web service endpoint*:

```
<wsdl:service name="ECRISService-v1.0">
   <wsdl:port name="tns:ECRISServicePort-v1.0" binding="tns:ECRISServiceBinding-v1.0">
      <soap:address location="http://plublic.host.at.stesta/ecris/service"/>
   </wsdl:port>
</wsdl:service>
<wsdl:service name="ECRISService-v1.1">
   <wsdl:port name="tns:ECRISServicePort-v1.1" binding="tns:ECRISServiceBinding-v1.1">
      <soap:address location="http://plublic.host.at.stesta/ecris/service"/>
   </wsdl:port>
</wsdl:service>
```

Example 25 – Using version numbers in the service endpoint definition

Versioning solution for ECRIS

As already described in the "Inception Report" document, it has been already identified in the NJR pilot project that the "big bang" roll-out approach, which requires all software systems of all partner Member States to be deployed simultaneously, does not work well when updates or changes have to be performed in the implementations. This is obvious in an environment such as the one of NJR and ECRIS where multiple partners, each with different schedules and constraints, need to be able to work on their software implementations without directly depending on the other partners.

It is thus necessary to define a versioning strategy that ensures that software implementations of the ECRIS technical specifications, which have not yet been upgraded to be compliant with the latest agreed changes, can still function correctly.

Please note that the term "compatibility" here refers to the technical compatibility of the artefacts that are commonly defined and exchanges between ECRIS applications, namely the *WSDL* files, the *XSD* files and the *XML* messages being transmitted between systems as well as the common reference tables used to facilitate the exchange of judicial information.

Versioning strategy

Overall, the following events in the versioning strategy are defined:

- **Minor backwards incompatible change(s):** for example a new restriction is added, minor changes are applied in the structure of a data type, a *web service* is altered by adding a new return message, or information is added in the common reference tables.
- **Major backwards incompatible change(s)**: for example massive changes in the *XML* structure, *web services* or *XML* data types.

Please note that the concept of "minor **compatible**" versions cannot be considered. Indeed, even if a small change keeps the compatibility on the level of the technical validation of the *XML* message against its *XSD* definition, it does not imply that the implementation that processes the data can still function properly without being adapted. Indeed, even minor changes such as making a mandatory element optional, adding a new record in a reference table or increasing the length of a text element may disrupt the functional and/or business processing of the information.

In view of those events and their lifeline, it is proposed to use the following versioning format:

X.Y

Each letter represents a positive integer number:

- X is the number of iterations in which **major incompatible** changes have occurred
- Y is the number of iterations in which **minor incompatible** changes have occurred

There is no limit to the number of changes performed per iteration. Also, multiple changes of various degrees can happen within the same iteration (for example 3 minor incompatible changes and 2 major incompatible changes). In such a case, the iteration number of the higher degree is only iterated, whereas the iteration number of the lesser degree is set to 0.

More concretely, assuming the ECRIS *Detailed Technical Specifications* are on version 1.**9** and 3 minor incompatible changes are decided (for example increasing the size of a text field, removing the size restriction all-together of another field and adding a new element) the new version that will be produced will be versioned as 1.**10**. However, if along the changes mentioned above were included also the addition of a totally new structure within a message as well as a deep change in the structure and kinematics of an existing functionality, the new version would then be tagged as **2.0**.

Thus, based on the example given above, the following explains the compatibility relation between versions:

Version 1.9 is **incompatible** with version 1.10, which is **incompatible** with version 2.0

The definition of the future versions of the ECRIS technical specifications, as well as all future changes to these specifications, will need to be agreed upon in judicial and technical workgroups involving all Member States experts, with proper coordination and communication to be performed by a central entity.

The versioning solution is based on the following rules:

- Each ECRIS implementation **must be able to support 2 versions** of the ECRIS technical specifications at any given time. This allows avoiding "big-bang" deployments and each Member State can upgrade its implementation independently from the other Member States.

  Please note that this rule does not impose keeping two versions running and operational if the previous version is no longer used by any Member State. Ideally, as soon as a previous version stops being actively used, the Member States experts should agree and coordinate its withdrawal within the ECRIS technical workgroups.

- In addition, all messages within a **functional** transaction must be sent in the same version of the specifications. The **functional** transaction is understood as being the complete sequence of interrelated messages that are sent asynchronously between two Member States for completing a business dialogue (for example a request and then the response to the request).

  Example: if Member State "A" sends a request to Member State "B" in version 1.4, even if "A" and "B" both already support v1.5, "B" must still send the response back in version v1.4. However new functional transactions must be sent in the newer version 1.5. This implies that the shift to a newer version is achieved only progressively between two ECRIS partners.

- Because of the progressive transition to newer versions described earlier, it is necessary to define a maximum duration for these functional transactions in the *ECRIS Detailed Technical Specifications*. If this is not done, there is a risk that upgrades to newer versions remain blocked because functional transactions remain pending too long.

- If a new version is available and implemented by two Member States, and as soon as messages have been sent in the newer version, starting new functional transactions (for example sending new requests or new notifications) in the older versions must not be allowed. The Member State implementation receiving a message of a new functional transaction in an old version must return a specific "SOAP Fault" error code (to be defined in the detailed technical specifications).

- As already described earlier, it is not allowed to remove or modify records in the common reference tables. For removals, it is foreseen to utilise expiration dates whereas modification of an existing value will always be performed by adding a new element with the modified value.

The fact that the ECRIS technical specifications are versioned and evolve in time implies a soft rule for the national implementations of the ECRIS technical specifications: the Member States implementations **should not** try to store the *XML* messages received as such but they should rather extract the information contained in the *XML* messages, store this information independently from the *XML* structure and discard the *XML* messages when the functional transaction is finished. When retransmitting data received earlier, the national implementations should be able to extract the information previously stored and transform it into the appropriate *XML* structure at that moment for sending it to other Member States.

Versioning strategy implementation on the Service Contract

The versioning strategy described above is implemented on the level of the *WSDL service contract* using the XSD/*XML* design principles described earlier in this document:

- Major and minor backwards incompatible version numbers are used in the names of the ***WSDL* and *XML schema*** files as well as in the ***WSDL* and *XML schema*** target namespaces.
- The *web service* interface name (aka "portType") is suffixed with the major and minor version numbers.
- All *XML* messages defined in the *WSDL* contract are suffixed with the major and minor version numbers.
- All *web service endpoints* (i.e. bindings) that are defined in the *WSDL* contract are suffixed with the major and minor version numbers.

No additional software components, services or kinematics are added in the ECRIS technical specifications in order to implement the versioning strategy. The implementation of the versioning indeed relies on the fact that the invoker of a *web service* knows before issuing the call which version and which *endpoint* he intends to use. The following UML sequence diagram depicts this implementation:

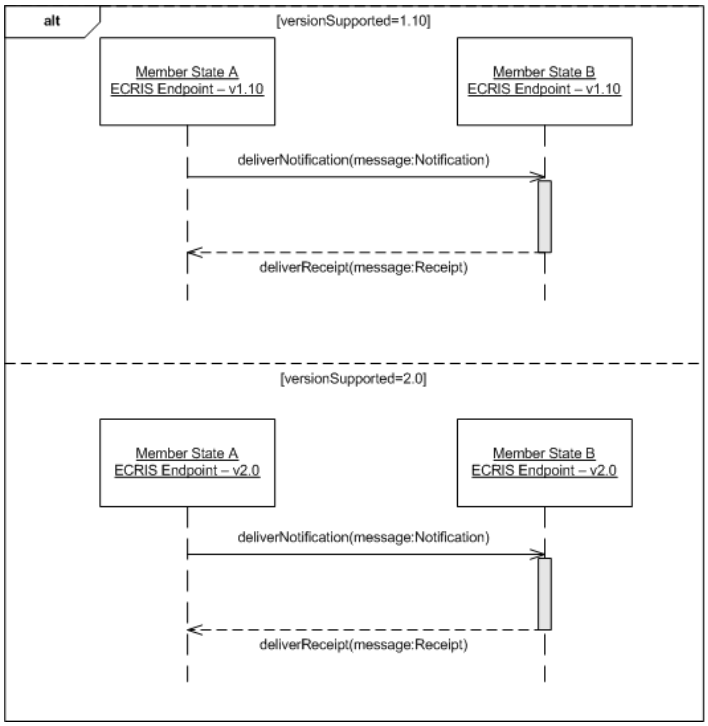


Figure 5 – "A" chooses the appropriate version and endpoint to use for sending a notification to "B"

In order to implement this strategy, each Member State implementation **must locally store** for all other Member States implementations the information of which *web service* end-point supports which version of the *ECRIS technical specifications* and at which *URL* address these *web service* end-points are available (i.e. this information is defined in the *WSDL* declarations of each *web services* implementation). In this way, each Member State implementation knows exactly where to send which version of the *XML* messages for each Member State.

The *WSDL* declarations of all Member States implementations will thus need to be managed by a central entity. In particular, when changing *web service* end-points, either by implementing new versions or by discarding previous versions, the Member States should first perform appropriate tests and then inform and transmit the modified *WSDL* declarations to the central entity. The central entity will then publish the *WSDL* declarations of all Member States and inform all stakeholders about the dates on which switches to newer versions will be done, when modified implementations will be deployed by the Member States, etc.

It needs to be underlined that when a Member State produces the *service contract* for its ECRIS implementation, special care has to be taken when filling in the "location" attribute of the <soap:address> element included in the <wsdl:service> definitions, since this will be the URL address that other Member States will use in order to send their *XML* messages. Thus, the URL address declared in this "location" attribute should be an address that is accessible from the other Member States via sTESTA rather than an internal network address.

Example: Versioning implementation in the *WSDL* contract

Let's assume that a minor version change has occurred in the ECRIS technical specifications so as to illustrate the effect that it will have in the *web service contract*.

Due to constraints applied from a business process, the allowed size of the property "RequestingAuthority" of type "xsd:string" has been restricted to 10 characters. After performing this modification, the new XSD file will look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified"
           targetNamespace="http://ec.europa.eu/ECRIS/messages-v1.6">
    <xs:complexType name="RequestMessageType">
        <xs:sequence>
            <xs:element name="RequestingAuthority">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:length value="10" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Example 26 – A new version of the *XML* schema is produced

Please note in bold the change of the minor incompatible version counter in the target namespace.

After that change, and in view of supporting at least two versions backwards, the *WSDL* will look like the following:

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xmlns:ecris-messages-v1.11="http://ec.europa.eu/ECRIS/messages-v1.11"
                  xmlns:ecris-messages-v1.12="http://ec.europa.eu/ECRIS/messages-v1.12"
                  xmlns:ecris-service-v1.11="http://ec.europa.eu/ECRIS/service-v1.11"
                  xmlns:ecris-service-v1.12="http://ec.europa.eu/ECRIS/service-v1.12"
                  xmlns:tns="http://ec.europa.eu/ECRIS/contract"
                 targetNamespace="http://ec.europa.eu/ECRIS/contract">
        <wsdl:types>
                <xs:schema targetNamespace="http://ec.europa.eu/ECRIS/service-v1.11" elementFormDefault="qualified">
                        <xs:import namespace="http://ec.europa.eu/ECRIS/messages-v1.11" schemaLocation="messages-v1.11.xsd" />
                        <!-- other definitions ommited for brevity-->
                </xs:schema>
                <xs:schema targetNamespace="http://ec.europa.eu/ECRIS/service-v1.12" elementFormDefault="qualified">
                        <xs:import namespace="http://ec.europa.eu/ECRIS/messages-v1.12" schemaLocation="messages-v1.12.xsd" />
                        <!-- other definitions ommited for brevity-->
                </xs:schema>
        </wsdl:types>
        <wsdl:message name="NotificationRequest-v1.11">
                <wsdl:part name="NotificationRequest" type="ecris-messages-v1.11:NotificationRequestType"/>
        </wsdl:message>
        <wsdl:message name="NotificationRequest-v1.12">
                <wsdl:part name="NotificationRequest" type="ecris-messages-v1.12:NotificationRequestType"/>
        </wsdl:message>
        <!-- other definitions ommited for brevity-->
        <wsdl:portType name="ECRISService-v1.11">
                <wsdl:operation name="deliverNotification">
                        <wsdl:input message="tns:NotificationRequest-v1.11"/>
                </wsdl:operation>
        </wsdl:portType>
        <wsdl:portType name="ECRISService-v1.12">
                <wsdl:operation name="deliverNotification">
                        <wsdl:input message="tns:NotificationRequest-v1.12"/>
                </wsdl:operation>
        </wsdl:portType>
        <wsdl:binding name="ECRISServiceBinding-v1.11" type="tns:ECRISService-v1.11">
                <!-- further binding configuration ommited for brevity-->
        </wsdl:binding>
        <wsdl:binding name="ECRISServiceBinding-v1.12" type="tns:ECRISService-v1.12">
                <!-- further binding configuration ommited for brevity-->
        </wsdl:binding>
        <wsdl:service name="SOAP_1.2_ECRISService-v1.11">
                <wsdl:port name="ECRISServicePort-v1.11" binding="tns:ECRISServiceBinding-v1.11">
                        <soap:address location="http://public.host.at.stesta/ecris/service"/>
                </wsdl:port>
        </wsdl:service>
        <wsdl:service name="SOAP_1.2_ECRISService-v1.12">
                <wsdl:port name="ECRISServicePort-v1.12" binding="tns:ECRISServiceBinding-v1.12">
                        <soap:address location="http://public.host.at.stesta/ecris/service"/>
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Example 27 – A *WSDL* declaration supporting two different versions

Without removing the definitions of the previous *web services*, a new version for each appropriate part (i.e. schema, portType, binding and service) was added so as to support the new minor incompatible version. *Web service* consumers who are not (yet) interested in the new version of the *web services* can still use the previous version for as long it remains available.

The same approach can also be used when a major incompatible change is performed. Also, the number of previous versions that are supported can be expanded if necessary and of course future versions can already be added for beta testing if required.

Tracking changes

Being able to identify the changes that have been performed through the lifecycle of an *XML* schema or of a *service contract* is quite critical. Given that *XML* is a descriptive language, tracking changes can be achieved easily by using comments within the *XML* schema or *service contract* definition. More concretely, comments are to be placed both in-line above the element that was changed, specifying the version within which the change took place, the actual date when it was performed, the contact e-mail address of the organisational entity that was responsible for performing the change in the technical artefact and a short description of what exactly was changed, as well as on the top of the file providing an overall description of the changes performed for each version. The implementation approach for tracking changes remains the same as with documentation: the <xs:documentation> element is to be used.

For example, let's assume that Mr. John Smith, following an ECRIS Expert Group meeting performed a change in the "commons-v1.0.xsd" *XML* schema and modified the length of the property "PersonName" of type "xsd:string" from 20 to 30 characters. The newly versioned "commons-v1.1.xsd" file would look like the following:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:commons="http://example.com/commons-v1.0"
                targetNamespace="http://example.com/commons-v1.0"
                elementFormDefault="qualified"
                attributeFormDefault="unqualified">
    <xs:annotation>
       <xs:documentation xml:lang="en">Date:01/04/2012 Contact:ecris-rep@ec.europa.eu Comment: Initial Version</xs:documentation>
       <xs:documentation xml:lang="en">Date:12/06/2012 Contact:ecris-rep@ec.europa.eu Comment: Updated by Mr. John Smith. Based on
the decision of the Expert Group on May 12th, 2012, the PersonName is changed from 20 to 30 characters so as to accommodate bigger
names</xs:documentation>
    </xs:annotation>
    <xs:complexType name="PersonType">
       <xs:sequence>
          <xs:element name="PersonName">
             <xs:simpleType>
                <xs:restriction base="xs:string">
                   <xs:annotation>
                      <xs:documentation xml:lang="en">Version :1.1 Date:12/06/2012 Contact:ecris-rep@ec.europa.eu Comment: Changed
size to 30 chars based on decision of Expert Group</xs:documentation>
                   </xs:annotation>
                   <xs:length value="30" />
                </xs:restriction>
             </xs:simpleType>
          </xs:element>
          <!-- Other elements ommited for brevity-->
    </xs:complexType>
</xs:schema>
```

Example 28 – Tracking changes with comments

Binary Attachments

The legislative framework of ECRIS foresees that optionally, Member States can send fingerprints as part of the personal information included in requests, notifications or information messages. Additionally, some Member States mentioned an interest for exchanging binary information such as scanned versions of judicial decisions, scans of identification tokens (for example ID cards or driver's license). Given that fingerprints and attachments are binaries, the same technical mechanism could be used for transmitting such content.

Regardless of the implementation particulars which are discussed later in this chapter, the functional context for binary attachments has to be set by defining first which binary types for attachments should be allowed and defining the appropriate limitations.

Fingerprints (NIST files)

In regards to fingerprints, during the preliminary analyses performed in the "Inception Phase" it appeared that the prominent formats for exchanging fingerprints information are based on the Interpol-implementation of ANSI/NIST-ITL 1-2000. Derivatives of this format are already used by other European projects such as in particular "EURODAC" and "PRÜM".

NIST files allow not only sending the actual binary images of the scanned fingerprints but also provides a set of place-holders based on *XML* for exchange of meta-data.

One of the main differences with other European projects foreseeing the exchanges of fingerprints is that in ECRIS, the fingerprints are an additional option next to the structured conviction data to be transmitted in the *XML* messages. The personal data of the individual must already necessarily be transmitted in this *XML* message and does not need to be included again in parallel into a NIST file. Furthermore, in ECRIS the fingerprints are considered solely for facilitating the identification of the individual to which a request or conviction information relates to rather than trying to identify several possible suspects.

It is assumed that the fingerprints to be exchanged optionally in ECRIS will be used for performing ten-prints against ten-prints matching during the identification process. Therefore, the focus lies on defining appropriate manners for transmitting primarily the binary ten-print image files. In particular, additional information such as minutiae records, latent print images, facial marks, any other biometric data and *XML* meta-data possibly embedded in the NIST files are considered optional.

As a result, the optional exchange of fingerprints in ECRIS is to be performed by joining NIST files as binary attachments to the *XML* messages to be exchanged between the Member States. The NIST file should primarily contain the ten-print fingerprint image and optionally the palm-print images (if available), as grey-scale images of a resolution of 500 dpi, encoded and compressed with the "Wavelet Scalar Quantization" algorithm (WSQ).

Please note that the definition of the detailed content of the NIST file is out of scope of the *ECRIS Technical Specifications* project. It is therefore recommended to apply the same standard for NIST files as the one that has been defined for the PRÜM project. The detailed definition of this standard can be found in the Council Decision 2008/616/JHA of 23 June 2008 on the implementation of Decision 2008/615/JHA on the stepping up of cross-border cooperation, particularly in combating terrorism and cross-border crime, more specifically in "CHAPTER 2: Exchange of dactyloscopic data (interface control document)" of the annex.

Binary attachments limitations

File types

The binary attachments are limited to NIST files only in the first version of the ECRIS technical specifications in order to keep the implementations as simple as possible, given the short timeframe available until April 2012. Furthermore, allowing other types of files would open the door to potential security problems (also in the case of PDF files) and additionally potentially raise data protection issues since the content of such files cannot be easily limited or automatically verified.

Please note that it is not technically possible to limit the type of binary attachments on the *service contract* level, at least not without using additional extensions to the *web service* specifications. Thus, each implementation of the ECRIS technical specifications that supports binary exchanges, either produced by the Member States or the *ECRIS Reference Implementation* to be provided by the European Commission, must specifically check the MIME type of the received binary attachment during the functional validation phase and before further processing the file. In the case that the file is not a NIST file, a specific error code – to be defined in the *ECRIS Detailed Technical Specifications* – must be returned to the sender.

Message size

The total size of a message, including the *XML* payload as well as all binary files attached, is limited to 10 MB.

Binary attachments implementation specification

The communication protocol defined earlier – *SOAP* 1.2 – provides already the necessary features so as to allow binary attachments. More specifically, two popular ways for sending binary attachments are supported: "SOAP with attachments" (SwA or also named "MIME for Web Services") and *MTOM*.

DIME, which is mentioned here for the sake of completeness, could be considered as well a reasonable approach but has been superseded by *MTOM*. Also, please note that "*Base64"* encoding has also been considered but deemed inappropriate, both due to the fact that the final "*Base64*" encoded binary ends up being, on average, 37% larger than the raw, non-encoded binary data as well as because the parser on the receiving side needs to know about the encoding so that it can decode the payload.

It is also necessary to mention here the interoperability issues that using binary attachments in SOAP messages can raise. In regards to the Java/J2EE platform, the following matrix provides an overview of the specifications that different platforms support (courtesy of http://wiki.apache.org, the complete matrix can be found under http://wiki.apache.org/ws/StackComparison)

Standard	Axis 1.x	Axis 2.x	CXF	IBM WAS 7.x	JBossWS	XFire (1.2)	Metro @ Glassfish	Oracle AS 10g
SOAP with Attachments	Y	Y	Y	Y	Y	N	Y	Y
DIME	Y	N	N	Y	N	N	N	Y
MTOM	N	Y	Y	Y	Y	Y	Y	Y

The .NET framework supports *MTOM*, *DIME* and "SOAP with attachments", at least from version 2.0 onwards.

PHP provides support only for "SOAP with attachments", in the form of an extension provided by the *PEAR* project.

Even though "SOAP with attachments" is widely adopted and used instead of using "*Base-64*" encoding for attachments, its design is somewhat flawed, since the binary attachment is not part of the *SOAP* message, making this approach similar in a lot of ways to just passing a URI for the binary data and leaving it up to the message processor to do the retrieval task. This can present difficulties both in terms of acknowledging that the binary attachment was properly received by the *web service endpoint* as well as in using extensions such as "WS-Security".

With that in mind, *MTOM* instead of "SOAP with attachments" is to be used for exchanging binary attachments in ECRIS. More specifically, the ECRIS implementations – both the software produced by Member States and the *ECRIS Reference Implementation* – will need to adhere **completely** to the following standard:

- SOAP Message Transmission Optimization Mechanism (W3C Recommendation 25 January 2005)
  http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/

  This version is the latest final version at the time of authoring this document.

Binary attachments exchange kinematics

Regardless of the technical standard followed, different approaches exist regarding the kinematics that can be used for actually sending and receiving binary attachments.

One very common approach in exchanging binary attachments is the "Push" approach, where the sender includes all attachments he wishes to send in one message. In the context of ECRIS, this means that with each notification, request or response to a request, binary attachments can be included right away along with the *XML* message (if available).

On the positive side, this means that the sender transmits in one go all data and is assured that it was received as intended, binary files included. Another benefit of this approach is its simplicity, since it does not require extra kinematics for information retrieval and verification of delivery.

On the negative side, using this approach means that each message is loaded with a large amount of data that the receiver may not be interested in (since not all Member States are interested in sending and/or receiving fingerprints). In addition to the concerns on the performance that this approach implies, it also implies that all Member States implementations need to support the binary attachment standard chosen and at least appropriately implement the receiving part of the "Push" approach in order to preserve interoperability. Additionally, when a large number of messages are exchanged special care needs to be taken so as not to overload the network and server resources of the sending and receiving parties.

Another usual approach for exchanging binary attachments used mostly when high volumes of binary data are to be exchanged or for interoperability reasons, is the "Pull" approach. With this approach, the sender transmits to the receiver a pre-agreed list of references to the binary attachments that he wishes to send instead of directly sending the binary files. In particular, this list includes metadata of the binary attachments that allow the receiver to identify for example the format, size, content and physical of the files. The receiver then, based on this list, uses an additional service provided by the sender so as to actually download the binaries when and if he is interested in receiving them.

This approach has exactly the opposite merits and flaws as compared to the "Push" approach. In particular, since the binary exchange becomes an ad-hoc functionality triggered only on explicit request, it must not necessarily be implemented, interoperability issues are reduced, development time and costs can be reduced for some of the Member States and the impacts on performance are limited. However, exactly for the same reason, the sender cannot be easily assured that the binary attachments are appropriately received. In order then to avoid that flaw, additional kinematics needs to be defined but increase the overall complexity of the solution.

In the context of ECRIS, several Member States are interested in systematically exchanging fingerprints while other Member States are not allowed by their national laws to even receive fingerprints electronically. Thus, a hybrid solution supporting both “Push” and “Pull” approaches is to be used in ECRIS.

Implementing the "Push-Pull" Approach

In order to implement the "Pull" part of the approach, the following additional operations are required in the *service contract*, used solely for the purpose of optionally retrieving the binary files specified within the *XML* message that is transmitted by the sender:

- An additional *web service* must allow a receiving Member State to request the NIST files from the Member State that has sent the first message when fingerprints are available.
- Another specific *web service* must then allow the initial Member State to push the NIST file asynchronously to the receiving Member State that has specifically requested them with the previous web service.

This also implies that appropriate *XML* elements are defined in the notification, request and response messages for carrying information on the binary file attachments. Such elements need to provide the following information for each binary file attached:

- A unique ID, used for retrieving the file from the sender. This ID can also be used for associating the attachment with other information *XML* elements if required.
- The MIME type of the attached binary file.
- The name of the binary file attached.

Optionally, the following elements can also be transmitted:

- The title of the file
- The file size in kilobytes
- A comment
- An MD5 hash of the file

Additional information can be added to this *XML* element if deemed necessary.

In addition to the web services and *XML* data elements, appropriate functional error codes must also be foreseen in the *ECRIS Detailed Technical Specifications* so as to communicate errors that might occur during this specific message exchange.

Concretely, if Member State "B" receives from Member State "A" a message indicating that binary attachments are available, "B" may at any point in time after receiving the message issue a new *web service* call for requesting one of the binary attachments that was indicated in the original message. "A" can then process the request asynchronously and "push" the binary file as a response to this request. The following UML sequence diagram further illustrates this message exchange:

Figure 6 – A successful binary attachment exchange between two Member States

In order to implement the "Push" part of the approach, for each operation defined in the *service contract* and that could potentially carry also binary attachments, such as "deliverNotification", "deliverRequest" or “deliverResponse”, a duplicate operation is defined suffixed with "Push" (i.e. "deliverNotificationPush", "deliverRequestPush", “deliverResponsePush” operations are added). The normal operation assumes that the “Pull” approach is used and implements it. The duplicate operation, suffixed with “Push”, behaves in the same way as the original operation but additionally supports receiving the binary attachments bundled together with the *XML* message.
By default, all software implementations use and implement the “Pull” approach. Implementations of Member States that do not support the operations suffixed with "Push" must return a specific “SOAP Fault” exception to be defined in the *ECRIS Detailed Technical Specifications* (for example "FunctionNotImplementedFault"), so as to notify the service caller that this functionality is not implemented.

As with the versioning implementation described earlier, no additional software components, services or kinematics are added in the ECRIS technical specifications to support usage of this approach. Indeed, the “Push-Pull” approach relies on the fact that the invoker of a *web service* knows before issuing the call which operation and which *endpoint* he intends to use. This implies that each Member State implementation implementing the “Push” part of the approach must store locally an information indicating for each other Member State whether the “Push” operations are supported or not (i.e. this information is defined in the *WSDL* declarations of each *web services* implementation).

The immediate benefit of this approach is that each Member State can freely decide and implement the most suitable approach. The ones not interested in actively participating in the binary attachments exchange do not need to implement the "Push" approach whereas Member States that are interested can opt for using this functionality when supported instead of using the "Pull" approach. Additionally, the impacts on performance can be limited and controlled, since Member States can chose to opt for the "Pull" approach if the volume of data to be exchanged is high (for example when sending a lot of messages within a short timeframe).

Please note however that Member States wanting to implement and use binary attachment exchanges using the "Push" approach will need to develop and deploy both “Push” and “Pull” solutions, thus increasing the cost of their development.

Batch transmission of Messages

During the "Inception Phase" of the *ECRIS Technical Specifications* project, it appeared that several Member States participating in the NJR pilot project are sending messages in batches and not in real-time. This is particularly done for notifications since these are rarely sent right away automatically when convictions are registered in the criminal records register. Indeed, new convictions and changes are usually piled up, then a small buffer period is used in order to make sure that the conviction information is no longer modified and then only the group of notifications is sent at once to the other Member States (for example once per day in the evening).

Please note that the NJR technical specification has not been designed to handle batches of *XML* messages, and each message is handled as a single autonomous unit. The obvious benefit of doing so is that the implementation remains simple and straightforward to implement.

In the first version of the ECRIS technical specifications, in order to keep the implementation as simple as possible and considering the short timeframe given until April 2012 for developing the ECRIS software, no batch operations allowing to send within one call several *XML* messages are to be defined in the ECRIS *web services*. Indeed, this would add supplementary work and complexity for the implementation of ECRIS. Furthermore, and since technical validation is to be done synchronously, if there is an error in one of the messages contained in the batch, then the whole batch is discarded. Additionally, receiving at once a lot of notifications is also not desired by the end users of ECRIS since they may not be able to treat a huge number of notifications at once.

However, in order to avoid problems in the processing of unexpected large numbers of messages, recommendations on the maximum number of messages per timeframe and per type will be provided in the *ECRIS Detailed Technical Specifications*, in order to establish a base of good practice regarding the transmission of messages.

Annex I – Overview of Member States Answers

Would you consider using a centralised architecture for the ECRIS information exchanges?

- Yes: AT, IT, LT, RO, SE and UK
- No: DE, EE, ES, FR, FI, LU, PL and SK
- Mixed: CZ
- Unknown: BE, NL, PT and SI

In particular, most of the Member States replying negatively consider that it would go against the ECRIS legal basis to use a centralised communication approach.

More specifically, the answers of the Member States in regards to the various questions concerning centralisation of technical artefacts are the following:

- Using a shared central host for referencing *WSDL* and *XML* files:
  - Yes: AT, BE, FR, IT, LT, NL, PL, SE, SK, UK
  - No: DE, EE, ES, LU
  - Unknown: CZ, FI, PT, RO, SI
- Using a shared central server for hosting common and/or national reference tables:
  - Yes: AT, BE, IT, LT, NL, PL, SE, UK
  - No: DE, EE, ES, LU, RO, SK
  - Mixed (yes for common, no for national tables): FR
  - Unknown: CZ, FI, PT, SI
- Using a central workflow system for handling transactional behaviour:
  - Yes: IT, LT, UK
  - No: AT, DE, EE, ES, FR, LU, NL, PL, SK
  - Unknown: BE, CZ, FI, PT, SE, SI, RO
- Using a central orchestration system of web services handling the kinematics of the data exchanges:
  - Yes: IT, LT, UK
  - No: AT, BE, DE, EE, ES, FR, LU, NL, PL, SK
  - Unknown: CZ, FI, PT, SE, SI, RO
- Using a central sequence number generator for establishing technical identifiers of messages:
  - Yes: IT, LT, UK
  - No: AT, BE, DE, EE, ES, FR, LU, NL, PL, SK
  - Unknown: CZ, FI, PT, SE, SI, RO

- Using a central server that would monitor the activities of the ECRIS servers in each Member State and indicate server health:
  - Yes: FR, IT, LT, UK
  - No: AT, BE, DE, EE, ES, LU, NL, PL, SK
  - Unknown: CZ, FI, PT, SE, SI, RO

The following table indicates the support of development platforms of the Member States for *web services* extensions:

	AT	BE	CZ	DE	EE	ES	FI	FR	LT	LU	LV	NL	PL	PT	RO	SE	SI	SK	UK
WS-Addressing	N	?	Y	N	Y	Y	N	?	?	N	?	?	Y	?	N	Y	?	Y	Y
WS-Discovery	N	?	Y	N	Y	N	N	N	?	N	?	?	N	?	N	Y	?	?	Y
WS-Security	N	?	Y	N	Y	Y	N	N	Y	N	?	?	Y	?	N	Y	?	Y	Y
WS-SecureConversation	N	?	Y	N	Y	Y	N	N	?	N	?	?	Y	?	N	Y	?	Y	Y
WS-Policy	N	?	Y	N	Y	N	N	N	?	N	?	?	Y	?	N	Y	?	Y	Y

ANNEX II – DISCARDED PROPOSALS

The following sections describe various technical proposals that have been discarded as a result of the feedback provided by the Member States experts through the written comments and during the Expert Sub Group meeting on 22 September 2010.
For each proposal, the original description is provided as well as the main reasoning that led to deem the proposal inappropriate.

Communication Architecture

Proposal: Peer Services for Monitoring Connectivity

Connectivity issues have been identified as being quite tricky to handle in the NJR pilot project. Given that ECRIS and NJR are software applications, the capability to react regarding connectivity problems is minimal since they cannot intervene on the *OSI Layer 3* problems. Furthermore, both systems use the same decentralised architecture, using multiple different networks through which the messages are channelled.
This chapter elaborates on the idea to provide automated tools allowing to monitor the connectivity. Based on the proposals made for the future NJR specifications v1.5, it is proposed to define a variant of the "deliverPing" service of NJR that would provide additional information facilitating troubleshooting. The additional information to be provided as part of the reply of the “deliverPing” function would be:

1. Status of peer (alive and working)
   This information is also proposed in the “deliverPing” message of NJR, in the form of an integer value.

2. Uptime

The “uptime” can help identifying the cause of unavailability of service and is useful for troubleshooting connectivity issues and also collecting statistics on these causes. Let us assume for example that a Member State’s system which previously had a working connection with another Member State’s system transmits a message but that the operation is not successful because the target host is unreachable. After some time, a new attempt for transmitting the message is performed by the sender and now the outcome is positive. In such a case, the sending Member State can query the proposed service and check the host's uptime period. If the host was up and running during the time the first unsuccessful request was made, the sending Member State can already identify that the problem was related to network issues and not to problems with the target host.

3. Connectivity status with other known peers

This information can also help identifying the cause of unavailability of service and is useful for troubleshooting connectivity issues. Indeed it can provide the requestor with insight as to whether other hosts are not available to the target system due to overall connectivity issues (for example sTESTA connection being down) or because of configuration issues.

4. Administrative messages (such as planned downtimes)

This information element can provide a standardised mean for communicating foreseeable losses of service due to a planned downtime or possible performance degradation due to maintenance activities.

Reason for discarding

The functions provided in this proposal are not considered essential for starting the operation of ECRIS in April 2012. Indeed, the first version of the ECRIS technical specifications is to be kept as simple as possible due to the short timeframe available for implementing ECRIS until April 2012. Organisational measures for ensuring efficient communication between all Member States, such as a central body that can for example notify all partners of planned downtimes or maintenance activities, are the preferred option for the first version of ECRIS.

However the idea to implement more elaborate automated monitoring mechanisms using peer services can be reconsidered in later versions of ECRIS.

Proposal: *XML* Document as Autonomous Information Chunks

The general approach in achieving autonomy in information chunks is to include all relevant information of a given element in an *XML* document as soon as this element is part of that document.

As a concrete example, let's assume that a list of buildings needs to be sent to a technical maintenance unit. A common *XML* schema has been agreed upon which defines the necessary data types, validation rules and predefined lists of possible values to be used. Let's assume further that the building definition named "BuildingType" has a property "BuildingCity" of type "CityType" and a property "BuildingProvince" of type "ProvinceType". For “CityType” and “ProvinceType”, a common list of possible values has also been defined as an enumeration in the *XML* schema.

In view of making the *XML* document autonomous, it should not contain only references to the specific values in the enumerations for “CityType” and “ProvinceType” but, for each building, it should contain the actual complete information set:

```
<?xml version="1.0" encoding="UTF-8"?>
<Buildings>
        <Building id="12343">
                <BuildingCity>
                        <CityName>Athens</CityName>
                        <!-- other elements of City structure ommited for brevity-->
                </BuildingCity>
                <BuildingProvince>
                        <ProvinceName>Attica</ProvinceName>
                        <!-- other elements of Provice structure ommited for brevity-->
                </BuildingProvince>
        </Building>
        <Building id="56789">
                <BuildingCity>
                        <CityName>Athens</CityName>
                        <!-- other elements of City structure ommited for brevity-->
                </BuildingCity>
                <BuildingProvince>
                        <ProvinceName>Peiraus</ProvinceName>
                        <!-- other elements of Provice structure ommited for brevity-->
                </BuildingProvince>
        </Building>
</Buildings>
```

Example 29 – Autonomous *XML* documents

This enables any consumer of this *XML* document, and not only the software system of the intended consumer, to easily read and understand the information, also without knowing or have direct access to the common information sets and definitions. Additionally, even if the enumerations of possible values change in time with the addition or removal of cities or provinces, the *XML* document still remains valid and no information migration or transformation is required.
In the example above, it becomes obvious that including all values in the *XML* document leads to information duplication and thus redundancy. It is possible to simplify this by adding internal references to the common elements and values within the *XML* document.
Expanding the previous example, let's add properties "BuildingCityReferrence" of type "CityReferrence" and "BuildingProviceReferrence" of type "ProvinceReferrenceType" to the "BuildingType" element. Please note that these new properties extend the "xs:IDREF" base type of the W3C *XML* Schema specification which is used so as to reference identifiers defined by the "xs:ID" base type of W3C *XML* Schema. These can then be used for linking information within the *XML* document as follows:

```
<Buildings>
	<Building id="12345">
		<BuildingCityReference>123</BuildingCityReference>
		<BuildingProvinceReference >123</BuildingProvinceReference >
	</Building>
	<Building id="56789">
		<BuildingCityReference>123</BuildingCityReference>
		<BuildingProvince>
			<ProvinceName>Peiraus</ProvinceName>
			<!-- Other elements of province structure ommited for brevity -->
		</BuildingProvince>
	</Building>
	<ReferredCities>
		<City id="123">
			<CityName>Athens</CityName>
			<!-- Other elements of city structure ommited for brevity -->
		</City>
	</ReferredCities>
	<ReferredProvinces>
		<Province id="123">
			<ProvinceName>Attica</ProvinceName>
			<!-- Other elements of province structure ommited for brevity -->
		</Province>
	</ReferredProvinces>
</Buildings>
```

Example 30 – Autonomous *XML* documents with internal reference elements

Please note that in this example above both manners of including the values directly in the element and linking to a reference are supported simultaneously within the same *XML* document. It is proposed to allow this in ECRIS.

Reason for discarding

The ECRIS technical specifications intend to set a common protocol of interexchange of information between Member States' computerised systems. In particular, the *XML* messages are to be processed by the ECRIS applications and not to be read and manipulated by human operators. To that end, including all reference values in a message is considered inappropriate as it will lead to additional functional checks and create redundancy of information. In particular, it also adds complexity in the case where the reference information duplicated in the *XML* message does not match the reference information that was defined in the common reference tables.
Given the aforementioned arguments, this proposal is considered inappropriate and has been discarded.

Versioning

Proposal: Schema versioning – keeping constant *XML* namespace values and using the "version" attribute

The *XML* Schema specification allows an optional "version" attribute on the schema declaration. The advantage of this approach is that it is easy to implement and is fully supported by the *XML* schema standard. The impact upon *XML* instances is fairly minimal since the namespace remains unchanged. There are two disadvantages with this approach:

- *XML* schema validation tools are not required to validate instances using the "version" attribute — the attribute is provided purely for documentation purposes and is not enforceable by *XML* parsers.
- Since *XML* parsers are not required to validate using the "version" attribute, additional custom processing (i.e. in addition to parsing and validation) is required to ensure that the expected schema version(s) are being referenced by the *XML* instance.

As an example, the following XSD uses the version attribute to define the version of the schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://
ec.europa.eu/ECRIS/common/" version="1.0">
        .
        .
        .
</xs:schema>
```

Example 31 – Usage of "version" attribute

Reason for discarding

The proposed usage of the “version” attribute is only relevant in the case where a third level of versions, in particular minor compatible versions, is to be supported. Given the fact that the concept of minor compatible versions cannot be considered for ECRIS, the usage of the “version” attribute is no longer necessary.

Proposal: Design for extensibility using <xs:any>

The *XML* Schema standard introduces “<xs:any>” as a wildcarding element for enabling future extensions to schemas in a well-defined manner. “<xsd:any>” includes a “namespace” attribute that either constrains or extends the range of elements that might appear within the wildcard. The “namespace” attribute can be set to any of the following:

- “##any” enables the use of elements from any namespace to extend the schema
- “##targetnamespace” restricts wildcards to the elements that appear within a “targetNamespace”
- “##other” makes it impossible to extend the schema using elements from the “targetNamespace”

The “processContents” attribute dictates how schema extensions should be validated by an *XML* parser:

- “strict” requires the parser to validate all schema extensions
- “skip” turns off validation for schema extensions
- “lax” validates elements from supported namespaces and ignores unknown or unexpected elements (most Web services specifications use lax)

The example below illustrates the use of “<xsd:any>” to enable an extensible definition of a type "name":

```
<xs:complexType name="name">
 <xs:sequence>
  <xs:element name="first" type="xs:string"/>
  <xs:element name="last" type="xs:string"/>
  <xs:any namespace="##any"
                processContents ="lax"
                minOccurs="0"
                maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
```

Example 32 – Usage of <xs:any>

The example above would enable the *XML* instance to add additional constructs after the last name (for example Mr. John Doe, John Doe Senior, etc.) while remaining valid based on the schema definition.

Reason for discarding

The proposed design principle is mainly used to accommodate forward compatibility, which is not part of the versioning strategy that has been defined in the "ECRIS Technical Architecture".
Furthermore, it appears that some *XML* parsers have difficulties for properly parsing and validating *XML* documents based on schemas which use the <xs:any> tag.
Given all the above, this proposal is considered inappropriate and has been discarded.

Binary Attachments

Proposal I: Image Files Only

It is proposed to limit the transmission of fingerprints to WSQ files containing the binary image files of the ten-prints, without additional metadata.
These should be grey-scale images of a resolution of 500 dpi, encoded and compressed with the "Wavelet Scalar Quantization" algorithm (WSQ).
In this proposal, the attachments are expected to be in the form of any number of *.WSQ files. The total size of all attachments must not exceed 5 Megabytes.

Reason for discarding

The NIST file format has been retained for the "ECRIS Technical Architecture", which is a format that in addition to the image files allows carrying meta-data.

Proposal II: Fingerprints + PDF Files

As an extension to proposal I, it is proposed to allow also PDF files in view of exchanging scanned documents such as:

- Identification tokens, more specifically scans of ID cards, passports, driver's licenses, social security cards, etc.
- Court Decisions and amendments to them

Overall, regardless of the amount of binary attachments to be included per ECRIS message or the implementation particulars, the maximum size of the sum of all attachments that can be sent in one message must not exceed 6 Megabytes.

Reason for discarding

Allowing support for binary files other than NIST files potentially increases the security risks which would require additional mitigation measures to be implemented. In addition, since the content of PDF files cannot easily be limited or automatically verified, the support of such files would potentially also raise data protection issues.

Proposal III: Fingerprints, PDF files, Image Files, Compressed Files and MS Office Files

As an extension to the proposal II above, it is proposed that the following additional file types can be added as binary attachments:

- Image file types : JPG, BMP, PNG, GIF
- Microsoft Office files : DOC, DOCX, XLS, XLSX
- Compressed files: ZIP, RAR, GZIP, 7Z, BZIP2, TAR

The maximum size of the sum of all attachments that can be sent in one message must not exceed 6 Megabytes.

Image files can replace or be used as an alternative for the scanned documents mentioned earlier. To that end, the functional requirements that were defined in proposal I for scanned documents remain the same. Additionally, there are no limitations or preferences imposed as to the choice of format, analysis of the image and similar parameters as long as the total maximum file size of the message is not violated.

Microsoft office files can replace or be used as an alternative specifically for judicial decisions and amendments. When used in this fashion, the same functional requirements as in proposal I are still applied. Also, office files can be used to exchange other types of information, in which case the sender is responsible both for the validity and integrity of the information. Also, the sender is responsible to adhere to all rules and regulations regarding information. No further limitations are applied as to the functionality that can be included in these Microsoft Office documents, as long as the total maximum file size of the message is not violated.

Compressed files can be used in order to exchange group of files that fall under the two previous categories mentioned or for any other reason. All functional requirements and limitations already defined for the above categories apply to these files as well. Furthermore, there are no limitations or preferences imposed as to the choice of format, compression ratio and similar parameters as long as the total maximum file size of the message is not violated.

Reason for discarding

This proposal is discarded for the same reasons as Proposal II: Fingerprints + PDF files.

Proposal IV: No Limitations

As an extension to proposal III, it is proposed to not limit in any way the file formats that can be transmitted as attachments to the *XML* messages.

Thus any binary file can be used as a binary attachment as long as the maximum size of the sum of all attachments does not exceed 6 Megabytes.

Reason for discarding

This proposal is discarded for the same reasons as Proposal II: Fingerprints + PDF files.

Binary Attachments Exchange Kinematics

Proposal I: Alternative "Pull" Approach

This proposal is actually a variation of proposal II presenting the "pull" approach. It differs in the sense that, instead of using an additional web service for retrieving the binary attachments, the sender provides an external facility (such as an FTP server, secured e-mail, etc.) from which the binary attachments can be retrieved by the receiver. In that approach, additional *XML* elements need to be added to the *XML* messages for indicating to the receiver where and how the attachment can be retrieved.

This approach carries all the benefits of the approach discussed in proposal II and additionally it removes the need to use a technical standard that can support binary exchanges through *web services*. On the other hand, this approach definitely increases the administrative cost for all the Member States that wish to provide fingerprints (i.e. additional technical facilities are required for keeping the binary attachments, security concerns, maintenance) and the complexity of the kinematics will also increase so as to ensure a successful exchange of the information.

Reason for discarding

The hybrid "Push-Pull" approach has been retained as a result of the positive feedback generally received from the Member States experts.

This proposal is thus no longer necessary.

Batch Processing of Message Exchanges

Proposal: Batch Processing of Message Exchanges

The proposal is to introduce batching capabilities for the exchanges of the XML messages in order to reduce the cost in machine resources that is required to currently handle the complete transmission of information between Member States.

The benefit provided by that approach becomes obvious using a similar example. Indeed, serialisation to and from *XML* is only performed once when many *XML* messages are sent at once. Also the establishing of the secured communication channel is only performed once.

To succeed in using batches for transmission, the following elements have to be taken under consideration and solved

- Message correlation: The term correlation refers to the logical connection between the two messages. It must be possible to relate a response or an error message to exactly one of the *XML* messages sent in the batch.
- Validation of the *XML* document message payload against the *XML* schema(s) that define its structure before sending the batch.

Message correlation is easily achieved by adding a unique identifier to all top elements in a given batch. For example, if a sender is posting a batch of notifications, each notification needs to have a unique identifier (which is already the case in the NJR specification).

Format validation is a good practice that must be performed regardless of the use of batches, since before transmitting a message the sender should at least be positive that the message will not be rejected due to *XML* validity errors.

Please note that using batches is considered here as an additional feature to using the single message transmissions. Both functions need to be available in the ECRIS service contract.

Please note also that if the "push" approach is chosen for sending binary file attachments, then the batch sending of *XML* messages is also not recommended due to concerns in the volume of data to be exchanged (or the number of *XML* messages in the batch needs to be strictly limited in order to control the volume but this again reduces the interest in the batch sending approach).

Finally, implementing a *web service* that performs sending or receiving of batch messages is fairly straightforward and simple. Indeed, the processing code that performs the sending or receiving of a single *XML* message can be completely reused and put in a loop for realising the batch sending or receiving. Thus it does not increase dramatically the complexity of the solution or the development and testing costs.

Example of a batch of notifications

Let us assume that a kinematic has been established in which a sender can include multiple notifications in one transmission and the receiver must reply either with a "ReceiptMessage" for acknowledging that the notification is processed successfully or with a "BusinessErrorMessage" in case of business validation errors.

Let's also assume that a sender wishes to transmit 20 notifications in one transmission to a receiver. The *XML* message payload created by the sender would look like the following:

```
<Notifications id="23143123123">
        <Notification id="123321">
                .
                .
        </Notification>
        <Notification id="321123">
                .
                .
        </Notification>
        .
        .
</Notifications>
```

Example 33 – *XML* message containing 20 notifications

Based on the kinematics briefly outlined above, the receiver would first need to send a receipt, notifying the sender that the batch message has been received successfully. The receipt would look like the following:

```
<ReceiptMessage>
        <InReplyTo>23143123123</InReplyTo>
        .
        .
</ReceiptMessage>
```

Example 34 – Receipt reply to batch message

Let us assume now that the specific notification with id "123321" could not be processed due to a business error, whereas notification with id "321123" was processed successfully. The payload of the message for the first notification would look like the following:

```
<BusinessErrorMessage>
        <InReplyTo>23143123123</InReplyTo>
        <RelatedTo>123321</RelatedTo>
        .
        .
</BusinessErrorMessage>
```

Example 35 – Business error reply to batch message

The message for the second notification would look like the following:

```
<BusinessSuccessMessage>
        <InReplyTo>23143123123</InReplyTo>
        <RelatedTo>123321</RelatedTo>
        .
        .
</BusinessSuccessMessage>
```

Example 36 – Business success reply to batch message

Of course, the same batching approach could be used to deliver also these receipt and error messages, following the same paradigm that was used for notifications.

Reason for discarding

In the first version of the ECRIS technical specifications, in order to keep the implementation as simple as possible and considering the short timeframe given until April 2012 for developing the ECRIS software, no batch operations allowing to send within one call several *XML* messages are to be defined in the ECRIS *web services*. Indeed, this would add supplementary work and complexity for the implementation of ECRIS. Furthermore, and since technical validation is to be done synchronously, if there is an error in one of the messages contained in the batch, then the whole batch is discarded. Additionally, receiving at once a lot of notifications is also not desired by the end users of ECRIS since they may not be able to treat a huge number of notifications at once.