NIEM NDR September 30, 2006

National Information Exchange Model
Naming and Design Rules and
Data Modeling Guidelines

Draft Version 0.3
September 30, 2006

Editor:

Webb Roberts, Georgia Institute of Technology

Contributors:

Abstract:

This document specifies the data model, XML artifacts, and XML data for use with the National Information Exchange Model.

Status:

This document is an early draft of a specification for NIEM-conformant XML components. This document is incomplete, and will undergo considerable revision before being approved for use.

Please make comments on this specification via email to niem-comments@lists.gatech.edu.

Table of Contents

1. Introduction 4

1.1. The NIEM Reference Architecture (In Brief) 4

1.2. Scope 5

1.3. Audience 5

1.4. Document Conventions 5

1.5. Syntax and Formatting 7

2. Key Concepts and Terminology 9

XML data may be referred to as a NIEM-conformant instance if it conforms to this specification 10

3. Guiding Principles 11

3.1. Specification Principles 11

3.2. Data Model Principles 11

3.3. Principles in the use of XML 12

4. Relation to Standards 15

4.1. XML 1.0 15

4.2. XML Namespaces 15

4.3. XML Schema 15

4.4. ISO 11179 15

5. Naming Rules 19

5.1. Usage of English 19

5.2. Characters in Names 19

5.3. Use of Acronyms and Abbreviations 19

5.4. Singular and Plural Forms 20

5.5. Character Case 20

6. Normalized Structure Design Rules 21

6.1. Structures Namespace 21

7. General Schema Design Rules 25

7.1. Mixed Content 25

7.2. Notations 25

7.3. Schema Document Element 25

7.4. Top-Level Constructs 26

7.5. Import of Namespaces 27

7.6. General Type Definitions 28

7.7. Simple Type Definitions 29

7.8. Complex Type Definitions 30

7.9. Element Definitions 31

7.10. Element and Attribute Definitions 32

7.11. Attribute Declarations 32

8. Annotation Design Rules 34

8.1. User Information ("documentation") Elements 34

8.2. Application Information ("appinfo") Elements 34

8.3. Documented Components 35

8.4. Types of Annotations in Reference Schemas 35

8.5. NIEM `appinfo` Namespace 36

9. Subset Schemas Design Rules 37
9.1. Schema Document Element 38
9.2. Annotations 38
9.3. Simple Type Definition 38
9.4. Simple Content Definition 38
9.5. Complex Type Definition 39
9.6. Attribute Declarations 39
9.7. Complex Content 40
9.8. Element Definition 41
10. Constraint Schema Design Rules 42
11. Extension Schema Design Rules 43
12. Document Schema Design Rules 44
13. Conformant Instance Rules 45
14. NIEM Data Modeling Guide 46
14.1. Overview of Data Modeling 46
14.2. Normalizing Element Use 51
14.3. Element Substitution 52
14.4. Roles 56
14.5. Associations 57
14.6. Metadata 61
14.7. Class Augmentation in NIEM 66
14.8. Using Non-NIEM XML Dialects with NIEM 69
Appendix A. Supporting Files 75
Appendix A.1. Schema for Structures Namespace 75
Appendix A.2. Schema for entity appinfo namespace 75
Appendix A.3. Schema for xml namespace 76
Appendix B. Normative Abbreviations 77
Appendix C. References 78
Appendix D. Revision History 79
Appendix E. Glossary 80
Appendix F. Notices 81

1. Introduction

This document specifies the National Information Exchange Model (NIEM), an information sharing framework based on the World Wide Web Consortium (W3C) eXtensible Markup Language (XML) Schema. In February 2005, the U.S. Departments of Justice (DoJ) and Homeland Security (DHS) signed a cooperative agreement to jointly develop the NIEM by leveraging and expanding the Global Justice XML Data Model (GJXDM) into multiple domains. The NIEM is a result of a combined government and industry effort to improve information interoperability and exchange within the U.S. at federal, state, tribal, and local levels of government.

The NIEM specifies a set of reusable information components for defining standard information exchange messages, transactions, and documents on a large scale across multiple communities of interest and lines of business. These reusable components are rendered in W3C XML Schema. The resulting schemas are available to government practitioners and developers at `http://niem.gov/`.

The W3C XML Schema standard was designed to enable information interoperability and sharing by providing a common, extensible language for describing data precisely. The constructs it defines are basic metadata building blocks – baseline data types and structural components. Users employ these building blocks to describe their own domain-oriented data semantics and structures. A reasonable set of rules and constraints governing what XML Schema constructs are allowed and how to use them (i.e. a framework) helps to ensure that resulting user data components can be reused and shared consistently. This enhances information interoperability.

The NIEM Naming and Design Rules specify principles and enforceable rules for NIEM data components and schemas. This document is a product of the NIEM Program Management Office. Audience

The primary audience for this document is the justice practitioners and developers who employ XML for information exchange and interoperability. The XML schemas rendered from the NIEM still offer schema designers much flexibility and freedom to extend types and create new properties to satisfy requirements at the local level. However, these rules are intended to establish and, more importantly, enforce a degree of standardization at the national level.

1.1. The NIEM Reference Architecture (In Brief)

The NIEM is a reference model of unconstrained components rendered in XML Schema. Associated with the NIEM schemas is an XML reference architecture that organizes and guides the employment of the various kinds of schemas that compose a NIEM information exchange. The XML reference architecture is a visual representation of the relationships between XML schemas for NIEM Information Exchange Package Documentation (IEPD) depicted in Figure 1. A NIEM IEPD is a set of artifacts that describe an Information Exchange Package, a standard message structure as defined by the Federal Enterprise Architecture Consolidated Reference Model Document. Refer to the Global JXDM Information Exchange Package Documentation Guidelines, Version 1.1 for a more detailed explanation of IEPDs and their contents.

Figure 1: The NIEM Reference Architecture

Error! Objects cannot be created from editing field codes.

There are generally four categories of XML schemas used to specify the instances of a particular NIEM information exchange:

- the NIEM schemas (or a subset thereof),
- a constraint schema,
- an extension schema, and
- a document schema.

The latter three schemas are optional. The only mandatory schema is the NIEM base schema or a correct subset of it (Subset schema derivation is defined in Section 9: Subset Schemas).The NIEM schemas may import code table schemas (or subsets) as needed. An optional *document* schema imports, re-uses, and organizes the components from the NIEM for the particular exchange. An optional *extension* schema may be used to add extended types and properties for components not contained in the NIEM,.

The document and extension schemas can be combined into a single schema and namespace, or can be broken out into separate schemas and corresponding namespaces. The user may decide the best way to organize components. If the extension components will be reused elsewhere, it may be more efficient to maintain them in a separate namespace, rather than including them in a document namespace.

The NIEM schemas are all inclusive and unconstrained. By creating a subset, the user can limit the components to only those he needs. Subsets can be created from the NIEM base schema and code table schemas as well. The basic principle for a subset is that an instance that validates against a correct subset schema will always validate against the full NIEM schema set. The user may also adjust cardinality constraints as desired within the subset schemas. Additional constraints can be handled in a *constraint* schema. A constraint schema may be derived from the subset schema, however, it can contain other constraints (for example, xsd:choice). The constraint schema provides a second *constraint validation* path that allows the user to reduce the possible set of correct XML instances independently from the NIEM schema or subset *conformance validation* path. This is done through multi-pass validation. A correctly constructed XML instance will validate through both the conformance and the constraint path.

1.2. Scope

This document is a specification for the NIEM 1.0. It is not intended to specify beyond the NIEM 1.0 release. The document addresses several issues:

- Definition of NIEM-conformant schemas
- Definition of NIEM-conformant reference schemas, on which schemas that are simply conformant are based
- Definition of subsetting methodology, through which conformant schemas are built from conformant reference schemas
- Naming of content to ensure understandability and reuse
- Documentation of content to ensure comprehension
- Definition of NIEM-conformant instances, which contain additional validation requirements, such as types associated with references and relationships.

This document does not address the following:

- A formal definition of the data model. Such a definition would focus on RDF and concepts not strictly required for interoperability. The document instead focuses on definition of schemas that work with the data model, to ensure translatability and interoperability.
- Definition of versioning. The NIEM distribution has a versioning mechanism in place, consisting of version numbers, with rules for what constitutes a "minor" or "major" change, and rules for inter-version compatibility. Such rules are not strictly required for peer-level interoperability, and will be added at a later stage.
- Definition of envelopes. This document does not define mechanisms related to the transport of NIEM-conformant data between two points.

This document is intended as a technical specification. It is not intended to be a tutorial.

1.3. Audience

The primary audience for this document is government practitioners and developers who employ XML for information exchange and interoperability. Such information exchanges may be between organizations or within organizations. The XML schemas rendered from the NIEM still offer schema designers much flexibility and freedom to extend types and create new properties to satisfy requirements at the local level. However, these rules are intended to establish and, more importantly, enforce a degree of standardization on a national level.

1.4. Document Conventions

1.4.1. Logical Quoting

This document uses "logical quoting", in which, when required, exact terms are placed within quotes, with supporting punctuation placed outside the quotes. For example, when discussing a string with the value of "an exact value", we do not quote it as "an exact value," or as "an exact value." For such cases, we would use "an exact value", or "an exact value". In these cases, punctuation is placed outside the quotes, instead of within the quotes, as it would be with traditional quoting.

1.4.2. RFC 2119 Terminology

Within normative content (rules and definitions), the key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in **[RFC2119]**.

1.4.3. References

This document relies on references to many outside documents. Such references are noted by bold, bracketed inline terms. For example a reference to RFC 2119 is shown as **[RFC2119]**. All reference documents are recorded in Appendix C: References.

1.4.4. XML Information Set Terminology

The following terms are used as defined by **[XMLInfoSet]**:

- Element parent
- Element child

  Note that the "child" of an element is a direct, immediate child. Children of an element, and their children, etc, will be referred to as "descendants" of that element.

- Document element

  The term "document element" is preferred over "root element".

- Attribute owner element

  An "owner element" is the element that possesses or contains the attribute.

- Attribute references
  The "references" value of an attribute is the list of elements referred to by the IDREFS or IDREF value of an attribute.

1.4.5. XML Schema Terminology

The terms “W3C XML Schema” and “XSD” are used throughout this document. They are considered synonymous; both refer to XML Schemas that conform to Parts 1 and 2 of the *W3C XML Schema Definition Language (XSD) Recommendations* (**[XMLSchemaStructures]** and **[XMLSchemaDatatypes]**).

The term "schema component" is defined by XSD. XML Schema contains specific definitions for various elements acting as particular types of schema components, including "model group definition schema component" and "Element declaration schema component". Such definitions are referred to, rather than restated.

1.4.6. Normative and Informative Content

The NIEM NDR includes a variety of content. Some text is normative (binding in implementations), while other content is informative, including supporting text and specific rationale for rules. Some conventions used within the document include:

[Definition: <*term*>]

A formal definition of a term. Definitions are normative.

[Principle <*number*>]

A guiding principle for the NIEM. The principles represent the requirements, concepts, and goals that have helped shape the NIEM. Principles are informative, but act as the basis on which the rules are defined.

[Rule <category><number>]

> A binding rule. The rules are normative. They should state how they bind the users. Most rules apply to conformant schemas (q.v.), while others apply to instances or reference schemas (q.v.).

The rules are categorized, to make indexing simpler. Categories for rules are as specified in Table 1: Rule Categories.

Table 1: Rule Categories

Rule short name	Meaning
ATD	Attribute Definition Rules
ATN	Attribute Naming Rules
CSR	Constraint Schema Rules
CTD	Complex Type Definition Rules
DOC	Documentation Rules
GNR	General Naming Rules: Broadly-applicable rules for naming entities.
GXS	General XML Schema Rules
IND	Instance Document Rules
SSR	Subset Schema Rules
STA	Standards: The NIEM's relation to standards, standards compliance, and interpretation of standards
STD	Simple Type Definition Rules
STR	Structures: The NIEM's use of specific structural conventions to represent non-hierarchical data and object order.

Rule identifiers that are deleted or recategorized will not be reused until a major release milestone is reached, at which point all identifiers may be reset.

1.5. Syntax and Formatting

Courier: All words appearing in courier font are values, objects, and keywords.

Italics: All words appearing in italics, when not titles or used for emphasis, are special terms with definitions appearing in this document.

Keywords: keywords reflect concepts or constructs expressed in the language of their source standard. Keywords have been given an identifying prefix to reflect their source. The following prefixes are used:

- `xsd:` represents W3C XML Schema Definition Language. Use of the prefix "`xsd`" in schemas and instances is not required.
- `xsi:` represents W3C XML Schema's XML Schema Instance namespace. Use of the prefix "`xsi`" in schemas and instances is not required.
- `structures:` represents the NIEM structures namespace. Use of the prefix "`structures`" in schemas and instances is not required.
- `appinfo`: represents the NIEM appinfo namespace. Use of the prefix "`appinfo`" in schemas is not required.

[Defintion: structures namespace]

> The structures namespace for NIEM is represented by the URI "`http://niem.gov/niem/structures/1.0`".

[Definition: appinfo namespace]

> The appinfo namespace for NIEM is represented by the URI "`http://niem.gov/niem/appinfo/1.0`".

Rules and supporting text may use Extended Backus-Naur Form (EBNF) notation as defined by **[XML]**.

See Appendix E: Glossary for additional term definitions.

2. Key Concepts and Terminology

[Definition: Appinfo Namespace]

[Definition: Structures Namespace]

[Definition: NIEM-conformant schema]

[Definition: NIEM-conformant reference schema]

[Definition: Documented Component]

[Definition: NIEM-conformant subset schemas]

[Definition: NIEM-compatible constraint schema]

[Definition: NIEM-conformant instance]

[Definition: NIEM Conformant Namespace]

[Definition: NIEM Compatible]

[Definition: Placeholder Schema]

[Definition: Documentation Schema]

[Definition: Extension Schema]

[Definition: Code Table]

[Definition: Reference Element]

[Definition: Fundamental Element]

[Definition: NIEM-conformant schema]

> The term *NIEM-conformant schema* SHALL be defined as an XML Schema that complies with the rules for NIEM-conformant schemas as defined by this specification.

Rationale

> This specification is primarily concerned with defining a particular type of schema that is designed to match the numerous requirements and principles specified in Section 3: Guiding Principles.

[Definition: NIEM-conformant reference schema]

> The term *NIEM-conformant reference schema* SHALL be defined as an XML Schema that complies with the rules for NIEM-conformant reference schemas as defined by this specification.

Rationale

> This specification separates reference schemas from non-reference schemas. Reference schemas are the fully-documented forms of schemas that contain all available content, to the largest available cardinality.

These reference schemas may act as the basis for subset schemas, which are not reference schemas, and which may apply certain constraints, restrictions, and narrowing of scope to the reference schema.

Also included in this specification is the concept of constraint schemas. Constraint schemas act in tandem with the reference schemas, and act to restrict content specified by the reference schemas. Constraint schemas need not be NIEM-conformant, as the content on which they act must also validate against conformant schemas. Such validation may be performed in stages, in agreement with the principle of multiple-pass validation.

[Definition NIEM-conformant instance]

> A *NIEM-conformant instance* is a document or data set that satisfies the rules for NIEM-conformant instances as specified in this document.

Rationale

3. XML data may be referred to as a NIEM-conformant instance if it conforms to this specification

3. Guiding Principles

Principles in this specification provide a foundation and explanations for the rules. The principles are not operationally enforceable. The rules are the normative and enforceable manifestation of the principles.

3.1. Specification Principles

Principles regarding what to specify, and what this document covers.

3.1.1. Minimal Specification

This specification should state what is required for interoperability, not all that could be specified. Certain decisions (such as normative XML comments) could create roadblocks for interoperability, making heavy demands on systems for very little gain. The goal is not standardization for standardization's sake. The goal is to maximize interoperability and reuse.

[Principle 1]

> This specification should specify what is necessary for interoperability, and no more.

3.1.2. Schema-Level Specification

This specification should try, as much as is possible, to specify schema-level content. This is a specification for schemas, and so should specify schemas. It should avoid specifying complex data models, or data dictionaries.

[Principle 2]

> This specification should focus on providing rules for specifying schemas.

3.1.3. Specificity and Conciseness

A rule should be as precise and specific as possible, to avoid broad, hard-to-modify rules. Putting multiple clauses in a rule makes it harder to modify. Using separate rules allows specifics conditions to be clearly stated.

[Principle 3]

> This specification should feature rules which are as specific, precise, and concise as possible.

3.2. Data Model Principles

The definition of the data model follows numerous guidelines. It is based upon actual data requirements gathered from a large number of exchanges in the justice domain, as well as a need to regularize data definitions to make them understandable and implementable.

[Principle 4]

> NIEM schemas and data instances are constructed in such a way as to maintain consistency of its fundamental data model.

3.2.1. RDF Data Model

The NIEM data model is defined with the RDF data model at its core. The rules specified in this document ensure that NIEM-conformant XML instances preserve the Subject-Property-Object triplets defined by RDF. This support will allow for leveraging of Semantic Web and other higher-level understanding of data.

[Principle 5]

> The NIEM data model follows the Subject-Property-Object data model defined by RDF.

The RDF data model is defined by **[RDFConcepts]**.

3.2.2. Specialization of Types

The NIEM embraces the fundamental concept of specialization of types. Through specialization, general concepts are made more precise for specific cases. Specialization of types involves the creation of new types by extending or restricting existing types.

[Principle 6]

Types are specialized through the use of derived types

3.2.3. Specialization of Properties

The specialization of properties involves basing narrow concepts on general concepts. Properties are described by **[RDFConcepts]** as characteristics or relationships. We represent them in XML as elements and attributes.

[Principle 7]

Properties are specialized through the use of derived properties

3.3. Principles in the use of XML

There are numerous methods and best practices for the use of XML.

3.3.1. Invariant Content

XML Schema has constructs that can make the data provided by XML processors different before and after schema processing. A sample of this is the use of attributes with default values. Before processing, there may be no attribute value, but after processing, the attribute value exists.

Within the NIEM, the process of validation of instances against schemas is solely validation: testing that data instances match desired constraints and guidelines. It should not be used to change the content of data instances.

[Principle 8]

The content of a data instance must not be modified by processing against schemas.

3.3.2. XML Schema for Validation

The NIEM is designed for W3C XML Schema validation. A primary goal is to maximize the amount of validation that may be performed by XML Schema validating parsers.

[Principle 10]

The NIEM should depend on W3C XML Schema validating parsers for validation of XML content.

XSD validates content using content models: descriptions of what elements and attributes may be contained within an element and what values are allowable. Mechanisms involving linking using attribute and element values are useful, but should only be relied upon when absolutely necessary.

3.3.3. Minimal implementation requirements

The NIEM is intended to be an open specification, supported by many diverse implementations. It was designed from data requirements and not from or for any particular system or implementation. Use of the NIEM should not depend on specific software, other than XML Schema validating parsers.

[Principle 11]

The NIEM should not depend on specific software packages, frameworks, or systems for interpretation of XML instances.

Similarly, the NIEM should be implementable with commercial off-the-shelf and free software products.

[Principle 12]

> The NIEM should be implementable with a variety of commercial off-the-shelf and free software products.

3.3.4. Reference Schema Defines Namespace Contents

The NIEM uses the concept of a *reference schema*, which defines the structure and content of a namespace. For each NIEM-conformant namespace, there is exactly one reference schema. A user may use a subset schema (q.v.) for a reference schema, but all instances must validate against a single reference schema for each namespace.

[Principle 13]

> Each NIEM-conformant namespace will be defined by exactly one reference schema.

3.3.5. Reuse of Namespaces

The NIEM is designed to maximize reuse of namespaces and the schemas that define them. When referring to a concept defined by the NIEM, users should ensure that instances and schemas refer to the namespace defined by the NIEM. User-defined namespaces should be used for specializations and extension of NIEM constructs, but should not be used when the NIEM structures are sufficient.

[Principle 14]

> NIEM-conformant instances and schemas should reuse the NIEM namespaces when possible.

Reuse is by reference to the namespace, with validation against reference schemas or reference subset schemas.

3.3.6. Specific Typing

As soon as an application has determined the name and namespace of an attribute or element used in NIEM-conformant instances, it will also know the type of that attribute or element. NIEM does not employ anonymous typing.

[Principle 15]

> Each attribute and element within the NIEM has a defined type.

3.3.7. Avoidance of Wildcards

Wildcards in schemas work in opposition to standardization. The effort of creating harmonized, standard schemas is to standardize a set of data. Wildcards allow non-standard data to be passed in otherwise standard messages. As such, users may receive non-standard data, and users may not be encouraged to extend in such a way that extensions may be distinguished from standardized content.

[Principle 16]

> Wildcards in standard schemas should be avoided

3.3.8. Schema Location as a Hint

[XMLSchemaStructures] specifies `schemaLocation`, an attribute of the `xsd:import` element of a schema, the `xsi:schemaLocation`, and `xsi:noNamespaceSchemaLocation` attributes of XML instances. In both of these uses, the specification explicitly maintains that the schema location specified is a hint, which may be overridden by applications. For example, from **[XMLSchemaStructures]**:

> The actual value of the `schemaLocation`, if present, gives a hint as to where a serialization of a schema document with declarations and definitions for that namespace (or none) may be found.

[Principle 17]

> Schema locations specified within NIEM-conformant reference schemas are hints, to provide default values to processing applications.

3.3.9. Multi-pass Validation

Systems that operate on XML data have the opportunity to perform multiple layers of processing. Data may be processed by middleware, XML libraries, XML Schemas, and application software.

[Principle 18]

> The primary purpose of XML Schema validation is to restrict processed data to that data that conforms to agreed-upon rules. This restriction is achieved by marking as invalid that data that does not conform to the rules defined by the schema.

The NIEM does not attempt to create a one-size-fits-all schema, to perform all validation. Instead, it creates a set of reference schemas, on which additional constraints may be placed. It also does not focus on language-binding XML Schema implementations, which convert XSD definitions into working programs. It is, instead, focused on normalizing language and preserving the meaning of data.

[Principle 19]

> Constraints on XML instances MAY be validated by multiple schema validation passes, using multiple schemas for a single namespace.

3.3.10. No Mixed Content

When validating XML instance data against W3C XML Schemas, mixed content is very difficult to constrain. Instances that use mixed content are difficult to specify, and complicate the task of data processing. Much of the payload carried by mixed content is unchecked, and does not facilitate data standardization or validation.

[Principle 20]

> NIEM-conformant schemas do not specify data that uses mixed content.

3.3.11. Application versus User Information

[Principle 21]

> XML data is primarily intended for automatic processing, not for human consumption.

XML should be made human-understandable whenever possible, but it is not targeted at human consumers. XML Schema is intended for validators and automatic processing. HTML is intended for browsers. Browsers and similar technology provide human interfaces to XML and other structured content. As such, structured XML content does not belong in places targeted towards human consumption. Human-targeted information should be of a form suitable for presentation.

3.3.12. Design for Extensibility

The NIEM is designed to be extended. Numerous methods are considered acceptable in creating extended and specialized components.

[Principle 22]

> The NIEM is intended for extension and augmentation by users and developers outside the standardization process.

4. Relation to Standards

The NIEM uses many public standards, and is influenced by many others. This section specifies to what specifications the NIEM conforms, and the specific rationale for differences from public standards.

4.1. XML 1.0

Artifacts of NIEM conform to the most recent recommendation for XML.

[Rule STA1]

NIEM-conformant schemas MUST conform to XML as specified by **[XML]**.

4.1.1. XML Comments

XML Comments are not schema constructs and are not specifically associated with any schema-based components. As such, comments are not considered semantically meaningful by NIEM, and may not be retained through processing of NIEM schemas.

[Rule STR18]

XML comments shall not be used for meaningful information about constructs within XML Schemas.

Rationale

Since XML comments are not associated with any specific XML Schema construct, there is no standard way to interpret comments. As such, comments should be reserved for internal use, and XML Schema annotations should be preferred for meaningful information about components.

4.2. XML Namespaces

[Rule STA2]

NIEM-conformant schemas MUST conform to the specification for namespaces in XML, as defined by **[XMLNamespaces]** and **[XMLNamespacesErrata]**.

4.3. XML Schema

The W3C XML Schema definition language has become the generally accepted schema language that is experiencing the most widespread adoption. Although other schema languages exist that offer their own advantages and disadvantages, the best approach is to base NIEM on W3C XML Schema.

[Rule STA3]

All NIEM-conformant schemas MUST be based on the W3C XML Schema Recommendations: XML Schema Part 1: Structures and XML Schema Part 2: Datatypes, as specified by **[XMLSchemaStructures]** and **[XMLSchemaDatatypes]**.

Rationale

This document is to be the specification for schemas and instances, not a specification for the specification itself. Those go in principles.

4.4. ISO 11179

4.4.1. ISO 11179, Part 4

4.4.1.1. Formulation of data definitions

The ISO 11179, Part 4, standard provides structure and rules for defining data definitions. The NIEM uses this standard with respect to summary definitions.

[Rule STA5]

> Within a NIEM-conformant schema, each XML element, attribute, and type definition SHALL follow the rules and recommendations of formulating data definitions given by ISO 11179, Part 4.

Rationale

> To advance the goal of creating semantically-rich NIEM conformant schemas, it is necessary that data definitions be descriptive and meaningful.

Note that NIEM definitions may contain extensive details about an XML element, attribute, or type, including such things as a rationale, examples, and domain-specific usages.

4.4.2. ISO 11179, Part 5

The ISO 11179, Part 5, standard provides a structure and rules for naming data elements. The NIEM uses this standard, with some specific refinements.

4.4.2.1. Object Class

In the NIEM, the *object class* that constitutes the first part of an entity name is interpreted as a real-world object class. That is, the object class term should reflect the real-world object classes and not specific data classes. It represents a real-world object rather than simply a collection of data.

4.4.2.2. Representation Terms

ISO 11179 part 5 requires the use of representation terms for data classes. The NIEM uses a specific set of representation terms.

[Rule GNR1]

> Each XML element, attribute, and type defined by NIEM-conformant schemas SHALL use a representation term from Table 2: Representation Terms unless the XML elements are of types with complex content.

Rationale

> A representation term defines the kind of value that is to be expected from the element. It is not needed for elements that are of types with complex content because they are comprised of other elements. There is no single kind of value to be expected within an element of complex content.

4.4.2.2.1. Types

[Rule GNR2]

> NIEM-conformant schemas SHALL use the representation term "Type" in the name of each non-enumerated XML Schema type.

Rationale

> Using the representation term "Type" immediately identifies XML types in a NIEM-conformant schema and prevents naming collisions with corresponding XML elements and attributes.

4.4.2.2.2. Simple Types

[Rule GNR10]

> The representation term "SimpleType" shall be used in the name of each XML Schema simple type.

Rationale

> Schemas are more comprehensible when referenced structures may be easily identified. Specific uses of simple types and complex types have similar syntax, but very different effects on data definitions.

Schemas that clearly identify complex types and simple types are easier to understand without tool support.

4.4.2.2.3. Code Types

[Rule GNR3]

NIEM-conformant schemas SHALL use the representation term “CodeType” in the name of each XML Schema type which:

1. is a complex type
2. has simple content, where
3. its simple content is an enumerated XML Schema type.

Rationale

Using the representation term "CodeType" immediately identifies XML Schema types in a NIEM-conformant schema that define code sets and prevents naming collisions with corresponding XML elements and attributes.

Table 2: Representation Terms

Representation Term	Definition
Amount	A number of monetary units specified in a currency where the unit of currency is explicit or implied.
BinaryObject	A set of finite-length sequences of binary octets.
Graphic	A diagram, graph, mathematical curves, or similar representation
Picture	A visual representation of a person, object, or scene
Sound	A representation for audio
Video	A motion picture representation; may include audio encoded within
Code	A character string (letters, figures or symbols) that for brevity, language independence, or precision, represents a definitive value of an attribute.
CodeText	A character string for which data values are codes but are not validated by the schema because there is no corresponding enumerated type present.
DateTime[1]	A particular point in the progression of time together with relevant supplementary information.
Date	A particular day, month, and year in the Gregorian calendar.
Time	A particular point in the progression of time within an unspecified 24 hour day.

[1] DateTime is not actually used in the NIEM reference distribution schema, but is available for use.

Representation Term	Definition
DescriptionText	A character string that is a description of a value, not the actual value itself.
Identifier[2]	A character string to identify and distinguish uniquely, one instance of an object in an identification scheme from all other objects in the same scheme together with relevant supplementary information.
Indicator	A list of two mutually exclusive Boolean values that express the only possible states of a Property.
Measure	A numeric value determined by measuring an object along with the specified unit of measure.
Numeric	Numeric information that is assigned or is determined by calculation, counting, or sequencing. It does not require a unit of quantity or unit of measure.
Value	A result of a calculation
Rate	A representation of a ratio where the two units are not included.
Percent	A representation of a ratio in which the two units are the same.
Quantity	A counted number of non-monetary units possibly including fractions.
Text	A character string (i.e. a finite sequence of characters) generally in the form of words of a language.
Name	A word or phrase that constitutes the distinctive designation of a person, place, thing or concept.
Type	The expression of the aggregation of properties to indicate the aggregation of lower leveled information entities. All Type names use this Representation Term

[2] "ID" (the abbreviation) is preferred over the full term "Identifier". It is indicated in the table of abbreviations.

5. Naming Rules

5.1. Usage of English

The English language has many spelling variations for the same word. For example, American English “program” has a corresponding British spelling “programme.” This variation has the potential to cause interoperability problems when exchanging XML components because of the different names used by the same elements. Providing a dictionary standard for spelling will mitigate this potential interoperability issue.

[Rule GNR4]

> NIEM information exchange XML elements, attributes and type names MUST be composed of words from the English language, using the prevalent U.S. spelling, as provided by the Oxford English Dictionary, Second Edition, 1989.

5.2. Characters in Names

[Rule GNR5]

> NIEM information exchange XML element, attribute and type names SHALL use only the characters from , in accordance with the use specified in that table.

Names of entities within the NIEM follow the rules of W3C XML Schema, by rule [Rule STA3]. Entities also must follow the rules specified for each type of XML Schema entity.

Table 3: Characters Allowed in Names

Character Title	Character Literal	Use
Letters		The first character of a name SHALL be a letter.
Uppercase letters	'A'-'Z'	The first character of the name of a type or an element SHALL be an uppercase letter
Lowercase letters	'a'-'z'	The first character of the name of an attribute SHALL be a lowercase letter
Digits	'0'-'9'	Digits SHALL NOT be used to enumerate. They may be used to specify a specific concept or standard.
Underscore	'_'	Underscores SHALL NOT be used in NIEM entity names
Hyphen	'-'	Hyphens may be used in the Representation Qualification Suffix portion of an element name
Period	'.'	Periods may be used to separate a property name from its Representation Qualification Suffix.

5.3. Use of Acronyms and Abbreviations

Acronyms and abbreviations can obscure meaning, and impair understanding and interoperability. They should be used with great care. Acronyms and abbreviations that are used must be documented, and used consistently.

[Rule GNR6]

A NIEM-conformant schema MUST consistently use approved acronyms, abbreviations, and word truncations within defined names. The approved shortened forms are defined in Appendix B: Normative Abbreviations.

Other acronyms and abbreviations will be used on a per-schema basis. Such abbreviations must be properly documented within the schema documentation.

[Rule DOC1]

[NIEM 3.1 CHANGE] A NIEM-conformant schema MUST specify ALL acronyms, abbreviations, and other word truncations within NIEM-conformant schema notation.

5.4. Singular and Plural Forms

[Rule GNR7]

Within NIEM-conformant schemas, element, attribute and type names MUST be in singular form unless the concept itself is plural.

The following is an example of correct name use:

```
PersonPhysicalFeature, PhysicalFeatureType
PersonPhysicalDetails, PersonPhysicalDetailsType
personNameInitialIndicator
```

5.5. Character Case

[Rule GNR8]

The upper camel case convention SHALL be used for naming elements and types.

Rationale

The use of upper camel case for names of types has become a defacto standard, to which NIEM conforms.

Examples of upper camel case names:

```
PersonName
JewelryStone
```

[Rule GNR9]

The names of attributes defined within NIEM-conformant schemas SHALL be formatted in lower camel case.

Examples of lower camel case names:

```
amountCurrencyCodeListVersionID
characterSetCode
```

6. Normalized Structure Design Rules

The NIEM enforces a regular structure on XML instances. NIEM provides a specific schema which contains base types for types in NIEM-conformant schemas. It provides base elements to act as heads for substitution groups. It also provides attributes that provide facilities not otherwise provided by XML Schema.

6.1. Structures Namespace

The NIEM provides a namespace containing structures for organizing data. These structures should be used to augment XML data. The structures provided are not meant to replace fundamental XML organization methods; they are intended to assist them.

[Rule STR2]

> The NIEM structures namespace shall be represented by the URI "`http://niem.gov/niem/structures/1.0`".

Rationale

> The structures namespace is a single namespace, separate from namespaces that define NIEM-conformant data. This document refers to this content via the prefix `structures`.

[Rule STR3]

> NIEM-conformant schemas and instances SHALL NOT use content within the NIEM structures namespace except as specified by this document.

Rationale

> It is an error to insert into the NIEM structures namespace types, elements, attributes, etc., that are not specified by this document.

6.1.1. Sequence ID

The attribute `structures:sequenceID` is provided to allow specification of sequential order of instances, when a complex type's defined element sequence is insufficient. A limitiation of XML Schema is that control of cardinality (the number of times an element may occur in an instance) requires the use of sequences of elements. This use of xsd:sequence defines the elements occurring within a type in a specific order. This order may not match the desired sequential order of the represented entities.

An example would be for proper names, where the natural order of the names may not appear in the same order as the sequence defined by a complex type. For example, some naming patterns have the family name as the last name of a person, while others have family name first, and others in the middle. Without a method for concretely define the desired sequence of the parts of a name, such data will be misrepresented in an XML instance.

The `sequenceID` attribute allows instances to express the sequential order of data relative to a parent. The order of data is as yielded by XSLT's sort element, with data-type of "number", and order of "ascending". Content with identical `sequenceID` values has undefined order.

[Rule STR4]

> The order of elements that are children of a NIEM-conformant element shall be presented as if their sequential order is as follows:
>
> 1. First, elements owning an attribute `structures:sequenceID`, in the order that would be yielded with their sequence IDs sorted via XSLT's sort element, with a data type of "number" and an order of "ascending".
>
> 2. Following those elements, the remaining elements, in the order in which they occur within the XML instance.

Rationale

Because of NIEM's use of structured, defined types, and its use of sequence, as well as various representation mechanisms, the order of data within an XML instance may require more precise definition. The true order of objects (such as parts of a name, or lines in an address, or parts of a phone number) may need an explicit method to define their order.

In this definition, the term "presented" may mean presentation to the user, reports, or transfer to other data systems.

[Rule STR6]

Within NIEM-conformant schemas and instances, the attribute `structures:sequenceID` SHALL NOT be interpreted as meaningful beyond an indicator of sequential order of an object relative to its siblings.

Rationale

Siblings of a data item are items that have the same parent. Note that, using the reference and relationships mechanisms, data objects may have multiple parents. The sequenceID is truly metadata, helping to express the structure of the data, rather than its content.

Note that reference elements have the same semantics as concrete data elements, and so follow the same rules for sequential order. By using reference elements, an entity may have one order within one structure, and another order within another structure.

Within NIEM-conformant schemas, the order of objects SHALL be given by sorting the objects by numerical value of their respective attribute `structures:sequenceID`, from smallest to highest. The relative order of objects with equal values for `structures:sequenceID` is their order within the XML instance. The order of objects with no value for `structures:sequenceID` is after all objects that have values for `structures:sequenceID`, in their relative order within the XML instance.is undefined.

6.1.2. References

In XML instances, the primary method of expressing relationships between data objects is by:

1. expressing the data objects as XML elements, and

2. having one element contain other elements

In this way, there is generally some implicit relationship between the outer element (the "containing" element, a.k.a. the parent element) and the inner elements (the "contained" elements, a.k.a. the child elements). Such expression of relationships is said to be by **containment**.

Expression of all relationships via element containment is not always possible. Situations that cause problems include:

- Circular relationships

  For example, if we say "Object1 has a relationship to Object2" and "Object2 has a relationship to Object1". Expressed via containment, this would result in *infinite recursive descent*.

- Repeated relationships

  For example, if we say "Object1 has a relationship to Object2" and "Object3 has a relationship to Object2". Expressed via containment, this would result in a duplicate of Object2.

A method that solves this problem is to use references. In a C or assembler, a pointer would be used. In C++, a reference might be used. In java, a reference value might be used. The method defined by the XML standard is the use of ID and IDREF. An ID refers to an IDREF. This is the method used by NIEM.

[Rule STR8]

Within a NIEM-conformant schema, a *reference element* is an element defined with a name of the form

NCName "`Reference`"

Where NCName is as defined by **[XMLNamespaces]**.

Rationale

Reference elements allow XML data to break free of the hierarchical data model, allowing reuse of data objects.

[Rule STR9]

Within a NIEM-conformant schema, a reference element SHALL be defined to be of type `structures:ReferenceType`. Any element of this type must be a reference element.

Rationale

Reference elements must be of the reference type, and elements of the reference type must be reference elements.

[Rule STR10]

Within a NIEM-conformant schema, element of the form

NCName1

and of the form

NCName1 "`Reference`"

(where the value of NCName1 is the same between the two forms) shall be defined to have identical semantics. The NIEM recognizes no difference in meaning between a reference element and an element that is not a reference element.

Rationale

NIEM-conformant data instances may use concrete data elements and reference elements as needed, to represent the meaning of the fundamental data. There is no implied difference in meaning between reference or concrete data representations. The two different methods are available for ease of representation. No change in meaning should be implied by the use of one method or the other.

Some parties assert that "included" data is intrinsic, while referenced elements are intrinsic. As applied to NIEM-conformant data, such assertions are in error.

[Rule STR13]

Within NIEM-conformant schemas, an element defined with a name not of the form defined in [Rule STR8] SHALL NOT be of type `structures:ReferenceType`.

Rationale

If an element is not named to be a reference element, then it may not be of reference type. Only reference elements may be of reference type.

The NIEM schemas define `structures:ReferenceType` to require the use of an attribute `structures:reference`, which is of type IDREF as specified by **[XML]**. According to the rules of XML, such an attribute must contain a value that is represented by an attribute of type ID. In NIEM-conformant instance, the targets of IDREFs are expected to be values of the attribute `structures:id`.

The NIEM schemas define `structures:ReferenceType` such that it is unavailable as a base for extension or restriction.

The NIEM schemas define `structures:ReferenceType` such that it has an optional attribute `structures:id`. This may be used to describe additional metadata or information about the relationship described by an element of type `structures:ReferenceType`.

[Rule STR16]

Within a NIEM-conformant instance, an element of type `structures:ReferenceType` MAY contain an occurrence of attribute `xml:id`.

Within a NIEM-conformant instance, the element referred to by an attribute `structures:reference` MUST be of a type valid for the object of the fundamental element of the reference element. This property is described by rules in the relevant sections.

7. General Schema Design Rules

The W3C XML Schema language provides many redundant features that allow a developer to represent a logical data model many different ways. Heterogeneous data models can become an interoperability problem in the absence of a comprehensive set of naming, definition, and declaration design rules.

This section establishes rules for XML schema elements, attributes, and type creation. Because the W3C XML specifications are flexible, comprehensive rules are needed to achieve a balance between establishing uniform schema design while still providing developers flexibility across the Justice and Public Safety domain.

7.1. Mixed Content

[Rule CTD1]

> The value of the attribute `mixed` within an element `xsd:complexType` or `xsd:complexContent` shall not have the value "`true`".

Rationale

> A NIEM-conformant schema does not define mixed content. NIEM does not support mixed content in XML elements. Exchange documents containing mixed content are difficult to process, define, and constrain.

External schemas may include mixed content, and may be used with NIEM via external adapter typs and external container elements.

7.2. Notations

Notations are not supported by the NIEM. Notations allow the attachment of system and public identifiers on fields of data.

[Rule GXS3]

> NIEM-conformant schemas SHALL NOT contain an occurrence of the element `xsd:notation`.

Rationale

> The notation mechanism is not supported by NIEM. The `xsd:notation` element defines a notation on a field of data.

[Rule GXS4]

> NIEM-conformant schemas SHALL NOT contain a reference to the type `xsd:notation`, or to a type derived from that type.

Rationale

> The notation mechanism is not supported by NIEM. The xsd:notation type defines a field to which system and public identifiers may be applied.

7.3. Schema Document Element

The features of W3C XML Schema allow for flexibility of use for many different and varied types of implementation. The NIEM NDR requires consistent use of these features. The document element of a schema is `xsd:schema`.

[Rule GXS18]

> In a NIEM-conformant schema, any occurrence of the element `xsd:schema` MUST own an attribute `targetNamespace`. The value of the attribute MUST match the production `<absolute-URI>` as defined by **[RFC3986]**.

Rationale

> Schemas without defined namespaces provide definitions that are ambiguous, in that they are not universally identifiable.
>
> Absolute URIs are the only universally meaningful URIs. Finding the target namespace using XML Base is overly complicated, and not specified by XSD. Relative URIs aren't universally identifiable, as they are context-specific.

The `xsd:schema` element contains an optional attribute `attributeFormDefault`. The value of this attribute is immaterial to a NIEM-conformant schema, as each attribute defined by a NIEM-conformant schema must be defined at the top-level, and so must be qualified with the target namespace of its declaration.

The `xsd:schema` element contains an optional attribute `elementFormDefault`. The value of this attribute is immaterial to a NIEM-conformant schema, as each element defined by a NIEM-conformant schema must be defined at the top-level, and so must be qualified with the target namespace of its declaration.

[Rule GXS21]

> In a NIEM-conformant schema, the element `xsd:schema` must own an attribute `version`, which must have a non-empty value.

Rationale

> It is very useful to be able to tell one version of a schema from another. Apart from the use of namespaces for versioning, it is sometimes necessary to release multiple versions of schema documents. Such use might include:
>
> - Subset schemas
> - Error corrections or bug-fixes
> - Documentation changes
> - Contact information updates
>
> In such cases, a different value for the `version` attribute implies a different version of the schema. No specific meaning is assigned to specific version identifiers.

7.4. Top-Level Constructs

Top-level constructs of a schema are those definitions which occur just below the `xsd:schema` element. This section considers such constructs that do not merit their own section.

7.4.1. Element `xsd:include`

Element xsd:include brings schemas defined in separate files into the current namespace. Its use can create difficulties with schema reuse, and increases the likelihood of conflicting definitions.

[Rule GXS9]

> A NIEM-conformant schema MUST NOT contain the element `xsd:include`.

Rationale

> Inclusion of namespaced schemas violates the principle that a single reference schema defines a namespace. It breaks a namespace up into arbitrary partial schemas, which needlessly complicates the schema structure. Inclusion of unnamespaced schemas complicates schema understanding as well, making it difficult to find the realization of a specific schema artifact.

7.4.2. Element `xsd:redefine`

The `xsd:redefine` element allows a schema to restrict and extend components from a namespace in that namespace. As described by **[XMLSCHEMA-1]**:

The definitions within the <redefine> element itself are restricted to be redefinitions of components from the <redefine>d schema document, *in terms of themselves*. That is,

- Type definitions must use themselves as their base type definition;
- Attribute group definitions and model group definitions must be supersets or subsets of their original definitions, either by including exactly one reference to themselves or by containing only (possibly restricted) components which appear in a corresponding way in their <redefine>d selves.

Such redefinition introduces duplication of definitions, as multiple definitions exist for components from a single namespace.

[Rule GXS10]

A NIEM-conformant schema MUST NOT contain the element `xsd:redefine`.

Rationale

Use of redefine provides an alternative definition for the contents of a namespace, in violation of the principle that a single reference schema defines a NIEM-conformant namespace.

7.5. Import of Namespaces

Namespaces used by a NIEM-conformant schema must be imported using the `xsd:import` element, in compliance with the XML Schema specification. Importing of namespaces is performed via the `xsd:import` element, which appears as an immediate child of the `xsd:schema` element.

[Rule GXS11]

Within a NIEM-conformant schema, any occurrence of the element `xsd:import` MUST own the attribute `namespace`. The value of the attribute MUST match the production `<absolute-URI>` as defined by **[RFC3986]**.

Rationale

An import that does not specify a namespace is enabling reference to non-namespaced components. NIEM requires that all components have a defined namespace. It is important that the namespace declared by a schema be universally defined, and unambiguous. XML Base processing is not specified by XML Schema, and so is not supported here.

[Rule GXS13]

Within a NIEM-conformant schema, any occurrence of the element `xsd:import` which imports a NIEM-conformant schema MUST own the attribute `schemaLocation`.

Rationale

An import that does not specify a schema location gives no clue to processing applications as to where to find an implementation of the namespace. Even though such a provided schema location may be overridden, it is important that an initial default be provided for processing.

[Rule GXS14]

In a NIEM-conformant schema, the value of any occurrence of the attribute `schemaLocation` owned by an element `xsd:import` MUST match either the production `<absolute-URI>,` or the definition of "*relative-path reference*", as defined by **[RFC3986]**.

Rationale

Default schemas must be provided for processing. These may specified either as absolute or relative URIs. Since URNs are not resolvable, they are inappropriate for use in `schemaLocation`. The requirement for conformance to "*relative-path reference*" is required to avoid the more obscure syntax of "*network-path reference*" and the system-specific "*absolute-path reference*".

[Rule GXS15]

In a NIEM-conformant schema, the value of any occurrence of the attribute `schemaLocation` owned by an element `xsd:import` MUST be resolvable to a XML schema document file that is valid according to **[XMLSchemaStructures]** and **[XMLSchemaDatatypes]**

Rationale

The object imported via xsd:import must be a schema document. The XSD spec requires that the "author warrants" that this is the case. This rule ensures that this is actually the case.

7.5.1. Importing Non-conformant Namespaces

Rules for schema locations are made more complicated by issues related to the importing of non-conformant namespaces. These issues include:

- NIEM may not redistribute the copyrighted work of others without permission
- Many non-conformant namespaces have no authoritative schema
- Many non-conformant namespaces have multiple schemas, representing different versions under the same namespace
- Non-conformant namespaces disagree as to what version of other namespaces they require

As a result of these issues, imports of non-conformant namespaces are not required to contain a `schemaLocation` attribute. To make this testable, imports of non-conformant namespaces are required to contain an appinfo element indicating that the namespace is not conformant.

[Rule GXS20]

Any element `xsd:import` that does not import a NIEM-conformant namespace MUST contain an `xsd:appinfo` annotation of the following form:

```
<appinfo:ConformantIndicator>false</appinfo:ConformantIndicator>
```

Rationale

This rule enables schema processors to determine if a namespace is conformant or not. If an import claims that a namespace is conformant, it may be easily verified. If the import indicates that the namespace is not conformant, the rules for non-conformant namespaces hold. It is an error to indicate that a NIEM-conformant namespace is non-conformant.

[Rule GXS16]

Within a NIEM-conformant reference schema, any occurrence of the element `xsd:import` that imports a non-conformant schema MUST have as an immediate child an occurrence of the element `xsd:annotation` which has an immediate child element `xsd:documentation`.

Rationale

Reference schemas must be properly documented. Conformant schemas are guaranteed to contain proper documentation and so need no additional documentation. Non-conformant schemas must be documented at the point of import, because such schemas do not follow NIEM documentation rules.

7.6. General Type Definitions

Since NIEM document and extension schema elements and types are intended to be reusable, all types must be named. This permits other types to establish elements that reference these types, and also supports the use of extensions for the purposes of versioning and customization.

The requirement that types be named is established by [Rule STD1 and [Rule CTD2].

NIEM-conformant schemas may not use xsd:anyType, because this feature permits the introduction of potentially unknown types into an XML instance. NIEM intends that all constructs within the instance be described by the schemas describing that instance – xsd:anyType tends to work counter to the requirements of interoperability. In consequence, particular attention is given to the need to enable meaningful validation of the NIEM document instances.

[Rule GXS17]

NIEM-conformant schemas SHALL NOT reference the type `xsd:anyType`.

Rationale

The type `xsd:anyType` provides a substantial wildcard by which untyped and unconstrained data may be carried. This violates several NIEM principles.

7.7. Simple Type Definitions

[Rule STD1]

Within a NIEM-conformant schema, any occurrence of the element `xsd:simpleType` MUST appear as an immediate child of the element `xsd:schema`.

Rationale

NIEM does not support anonymous / unnamed types in conformant schemas. All "top-level" types are required by XSD to be named, and are therefore globally reusable.

[Rule STD2]

Within NIEM-conformant schemas, any occurrence of the element `xsd:simpleType` MUST have an occurrence of the element `xsd:restriction` as an immediate child.

Rationale

Any simple type must be a restriction of another type. One alternative is "list", in which the resulting type is a list of entries, which should be structured via explicit XML, and not composed fields. The other alternative is "union", which combines different-typed entries into a single type, obscuring meaning of instance values.

[Rule STD3]

Within a NIEM-conformant reference schema, any occurrence of the element `xsd:simpleType` MUST have an occurrence of the element `xsd:annotation` as an immediate child.

Rationale

Reference schemas must be properly documented.

[Rule STD4]

Within a NIEM-conformant schema, any occurrence of the element `xsd:restriction` that is an immediate child of an element `xsd:simpleType` MUST contain an attribute `base`.

Rationale

All restrictions must restrict named types. NIEM does not support anonymous types.

Taking into account the other rules, this rule may be derivable, however, it is useful to have the point stand on its own.

[Rule STD5]

Within a NIEM-conformant schema, the value of the attribute `base` owned by an element `xsd:restriction` acting as part of a simple type declaration schema component MUST have a value that refers to a simple type defined by the XML Schema specification, or a simple type defined by a NIEM-conformant schema.

The content of the simple type definition then may add facets to the base simple type, in line with XSD specifications.

[Rule STD6]

Within a NIEM-conformant schema, the value of the attribute `base` owned by an element `xsd:restriction` acting as part of a simple type declaration schema component MUST NOT have a value that refers to `xsd:anySimpleType`

Rationale

`xsd:anySimpleType` is insufficiently constrained to provide a meaningful starting point for content definitions.

7.8. Complex Type Definitions

[Rule CTD2]

Within NIEM-conformant schemas, any occurrence of the element `xsd:complexType` MUST appear as a child of the element `xsd:schema`.

Rationale

NIEM does not support anonymous / unnamed types in conformant schemas. All "top-level" types are required by XSD to be named, and are therefore globally reusable.

[Rule CTD3]

Within NIEM-conformant schemas, an occurrence of the element `xsd:complexType` MUST NOT include the attribute `mixed` with a value of "`true`" or "`1`".

Rationale

NIEM does not support mixed content.

NIEM supports use of attributes and attribute groups.

7.8.1. Complex Content

Within `xsd:complexType`, NIEM supports use of `abstract`, `block`, and `final`. NIEM supports use of simple content, complex content, `xsd:choice`, groups, sequences, attributes, and attribute groups.

[Rule CTD4]

Within a NIEM-conformant schema, the element `xsd:all` SHALL NOT occur.

Rationale

NIEM does not support use of `xsd:all`. Use of concretely-sequenced elements within complex types simplifies many types of processing, and allows reference schemas to act as a base for highly constrained, yet interoperable, subset schemas.

[Rule CTD5]

Within a NIEM-conformant schema, an occurrence of the element `xsd:group` acting as a particle schema component according to **[XMLSchemaStructures]** MUST have values of "`1`" for the attributes `minOccurs` and `maxOccurs`.

Rationale

Cardinality is restricted to maintain the simple-sequence compatibility of complex content. NIEM does not permit complicated patterns of interlacing of elements. Elements have a strict sequential occurrence.

The value of "1" for minOccurs and maxOccurs is provided as default by XSD, and so need not be explicitly expressed.

[Rule CTD6]

> Within a NIEM-conformant schema, an occurrence of the element `xsd:choice` MUST have values of "`1`" for the attribute `minOccurs` and `maxOccurs`. This value may be implicit.

The value of "1" for minOccurs and maxOccurs is provided as default by XSD, and so need not be explicitly expressed.

[Rule CTD7]

> Within a NIEM-conformant schema, an occurrence of the element `xsd:sequence` MUST have values of "`1`" for the attributes `minOccurs` and `maxOccurs`. This value may be implicit.

The value of "1" for minOccurs and maxOccurs is provided as default by XSD, and so need not be explicitly expressed.

[Rule CTD8]

> Within a NIEM-conformant schema, complex content SHALL NOT declare occurrences of a single element using more than one element statement.

Rationale

> The goal here is simple sequences of elements. Allowing multiple element statements for a single element creates situations where "Foo" is followed by "Bar" and again by "Foo", which puts structural and organizational constraints within the XML data file.

[Rule CTD9]

> Within a NIEM-conformant schema, an element or attribute that is eliminated through restriction and reinserted by extension MUST conform to the original definition.

Rationale

> The derived and extended content must maintain the "is-a" nature of derivation: Derived type "Foo" is-a base type "Bar". Any constraints on "Bar" must be maintained in the derived type "Foo".

7.8.2. Exclusion of Wildcards

[Rule CTD10]

> NIEM-conformant schemas SHALL NOT contain an occurrence of the element `xsd:anyAttribute`.

The element `xsd:anyAttribute` may appear within constraint schemas.

[Rule CTD11]

> NIEM-conformant schemas SHALL NOT contain an occurrence of the element `xsd:any`.

Rationale

> The elements xsd:anyAttribute and xsd:any provide wildcards, which may carry undefined content, in violation of the principle of avoidance of wildcards.

The element `xsd:any` may appear within constraint schemas.

7.9. Element Definitions

[Rule NEWRULE]

> An element declaration schema component defined by a NIEM-conformant schema may have a type attribute indicating a NIEM-conformant complex type.

[Rule NEWRULE]

> An element declaration schema component defined by a NIEM-conformant schema may have an attribute value of true and a type definition of the XML Schema ur-type.

[Rule NEWRULE]

An element declara

7.10. Element and Attribute Definitions

[Rule ATN1]

Each XML element and attribute name defined by the NIEM MUST correspond to a single representation type.

Rationale

The name of a XML element or attribute from a NIEM-conformant schema should be concrete. The element or attribute name alone should be sufficient in determining not only the semantic meaning, but also the type structure of that element or attribute.

7.10.1. Specific Typing

[Rule ATD1]

NIEM-conformant schemas SHALL NOT declare attributes or elements to be of type `xsd:anyType` or `xsd:anySimpleType`,

Rationale

In accordance with the principle of avoidance of wildcards, NIEM schemas should not be able to pass untyped content. All content should have a comprehensible set of values that can be parsed. The type `xsd:anyType` allows untyped XML content to be carried as a payload. The type `xsd:anySimpleType` is a union of all possible simple types, and so provides no purposeful constraint on payload content.

7.11. Attribute Declarations

7.11.1. Global Attributes

The NIEM distribution features attributes that are common to all elements. These common attributes are declared as attribute groups and utilize the following rule.

[Rule ATD2]

If a Schema Expression contains one or more common attributes that apply to all elements contained or included or imported therein, the common attributes SHOULD be declared as part of a global attribute group.

Rationale

For example: see the Global JXDM global attribute group named ”SuperTypeMetadata”

7.11.2. Consistency of Attribute Content

[Rule ATD3]

NIEM-conformant schemas MUST NOT use the `default` attribute of the `xsd:attribute` element.

Rationale

The default attribute is used in conjunction with optional elements in attribute declarations. It provides a value for the attribute if the attribute does not appear. Such values are yielded to XML instance processing applications after schema validation occurs. The use of this attribute causes data presented to applications to be different than the data that appears in the instances themselves, in violation of the principle of invariant content.

[Rule ATD4]

NIEM-conformant schemas MUST NOT use the `fixed` attribute of the `xsd:attribute` element, except when used in conjunction with the `use` attribute having the value `"required".`

Rationale

The fixed attribute is used to ensure that a used attribute always has a specific value. When applied to an optional element, it acts like the default attribute, changing the content of the attribute upon schema validation. Using it with required attributes ensures that valid content always has the specific value, while allowing the pre- and post-validated content to be identical.

8. Annotation Design Rules

All NIEM-conformant schemas must include documentation. Some documentation is intended to be human readable ("user information"), and other documentation is machine-readable ("application information"). These terms come from **[XMLSchemaStructures]**, a normative source.

[Rule DOC10]

The document element `xsd:schema` must follow the rules for documented components.

Rationale

A schema creates a new construct (a namespace), which must be documented. Such documentation describes the namespace as a whole.

[Rule DOC11[

The document element `xsd:schema` must claim to be conformant using the appinfo element `i:ConformantIndicator`.

The i:ConformantIndicator element is the method used by NIEM-conformant schemas to indicate that they are, in fact NIEM-conformant.

8.1. User Information ("documentation") Elements

[Rule DOC3]

Within NIEM-conformant schemas, the content of `xsd:documentation` elements SHALL NOT contain structured XML data.

Rationale

According to the XSD specification the content of xsd:documentation elements is intended for human consumption. XML content is intended for machine consumption. As such, any XML content appearing in xsd:documentation should be in the context of human-targeted examples, and should be escaped using &lt and &gt.

See **[SchemaForXMLSchema]**, the schema for XML Schema, as an example.

[Rule DOC4]

The attribute `xml:lang` SHALL be used to indicate the language of user information in NIEM-conformant schemas.

Rationale

XSD spec indicates that user info should use xml:lang to indicate the language of the user info. Note that the value of xml:lang is inherited by child elements, so the attribute need not be owned directly by the xsd:documentation element.

8.2. Application Information ("appinfo") Elements

[Rule DOC5]

An `xsd:appinfo` element SHALL contain well-formed XML data that conforms to **[XMLNamespaces]**.

Rationale

Application information elements are intended for "automatic processing", and so should contain machine-oriented data, XML. Such XML should conform to specifications.[3]

[3] The XML Schema specification states "{user information} is intended for human consumption, {application information} for automatic processing."

[Rule DOC6]

Any element that is an immediate child of an `xsd:appinfo` elements SHALL be in a namespace.

Rationale

Appinfo may contain XHTML data (which has no schema), or NIEM appinfo data (which has a schema). Use of default namespace is OK, but content has to have a real namespace. The XML namespaces specification includes the concept of non-namespaced content. Non-namespaced data confounds the concept of distinctly identifiable data definitions.

[Rule DOC7]

Within a NIEM-conformant reference schema, a namespace that is a descendent of an `xsd:appinfo` element SHALL be imported using the `xsd:import` element.

Rationale

The import of appinfo content is not strictly required by the XSD specification, but some tools break without it, and it helps users maintain connections between namespaces and implementations.

8.3. Documented Components

There are many types of components within a NIEM schema. Many of these components have identical rules regarding techniques for documentation. The rules in this section apply when a rule for a component type indicates that the component is a documented component.

[Definition: Documented Component]

A documented component is any component defined by a NIEM-conformant schema which requires documentation. Documented components are indicated as such by component-specific rules.

[Rule DOC9]

A documented component must contain a definition. Its definition is the first occurrence of an element `xsd:documentation` that is a child of an element `xsd:annotation` that is a child of the element that defines the component.

8.4. Types of Annotations in Reference Schemas

8.4.1. xsd:documentation: Summary

In keeping with **[XMLSchemaDatatypes]**, the content of xsd:documentation elements is intended for human consumption, not machine consumption. As such, it should contain text, not XML, except when the intent is to provide XML examples. In such cases, the escape sequences "&lt;" and "&gt;" should be substituted for the XML brackets "<" and ">" respectively.

[Definition: summary documentation element]

Within a NIEM-conformant reference schema, a *summary documentation element* SHALL be defined as an element `xsd:documentation` which does not own an attribute `structures:annotationCategoryURI`.

Rationale:

Any documentation element which does not carry an annotationCategoryURI attribute is assumed to be a summary.

[Rule DOC8]

Within a NIEM conformant reference schema, there SHALL exist a summary documentation element as a child of an element `xsd:annotation` that is a child of every occurrence of the following elements: `xsd:import, xsd:simpleType, xsd:complexType, xsd:group` when acting as a model group definition schema component, `xsd:attributeGroup` when acting as an attribute group definition schema component, `xsd:element` when acting as an element declaration schema component, or `xsd:attribute` when acting as an attribute declaration schema component.

Rationale

These elements are the elements that act as definitions. They must be annotated properly, including basic summaries. Note that the specific "acting" clauses are clearly defined in the XSD specification.

8.4.2. xsd:documentation: Full Description

8.4.3. xsd:appinfo: For Components

8.4.4. xsd:appinfo: List of Abbreviations

8.5. NIEM `appinfo` Namespace

To enable higher-level constructs beyond those provided by XML Schema, the NIEM includes additional, non-schema values to provide information about constructs in schemas. These properties are represented by elements from a specific namespace, referred to as the `appinfo` namespace. The `appinfo` namespace for NIEM is "`http://niem.gov/niem/appinfo/1.0`". The schema for this namespace defines several elements that are used in NIEM schemas.

8.5.1. The `ConformantIndicator` element

The element `appinfo:ConformantIndicator` is used for two purposes.

1. To indicate that a schema is conformant, or represents a conformant namespace.

2. To indicate that an imported schema is not conformant, or represents a non-conformant namespace.

[Rule NEWRULE]

The element `appinfo:ConformantIndicator` shall have a value of either true or false. The element MUST appear in appinfo for a component and will indicate:

- For a schema component, it indicates:
  - `true`: the schema is a NIEM-conformant schema
  - `false`: the schema is not a NIEM-conformant schema
- For a import element, it indicates:
  - `true`: the imported schema represents a NIEM-conformant namespace
  - `false`: the imported schema does not represent a NIEM-conformant namespace

9. Subset Schemas Design Rules

A subset schema is a NIEM-conformant schema which is derived from a NIEM-conformant reference schema. The primary rule is that any instance that validates to the subset schema must validate to the reference schema.

Note that these rules are not intended to act as a guide or procedure for generating subset schemas from reference schemas. They are intended to act as a set of constraints that ensure that generated schemas are properly defined subsets.

[Definition: NIEM-conformant subset schema]

> A *NIEM-conformant subset schema* is defined as a NIEM-conformant schema which is derived from a NIEM-conformant reference schema according to the rules provided by this document.

Rationale

> A subset schema is as defined by this document.

[Rule SSR2]

> A NIEM-conformant subset schema MUST be constructed such that any instance that validates against the subset schema SHALL validate to the reference schema on which it is based. All other rules regarding subset schemas are designed to support this rule.

Rationale

> The most important rule regarding subset schemas is that they are to be transparent to the validating application. Any instance that validates to the subset schema must be able to validate against the reference schema. In this way, the subset schema is a schema for documents that contain a subset of the content available to documents that validate against the reference schema.

[Rule SSR3]

> A NIEM-conformant subset schema SHALL be derived only via transformations explicitly allowed by this document.

Rationale

> This document describes all of the transformations available to produce subset schemas. Other transformations do not result in valid subsets. If additional transformations are discovered, they should be added to this specification.

[Rule SSR4]

> A NIEM-conformant subset schema SHALL be composed of the content of the NIEM-conformant reference schema, modified by transformations allowed by this document.

Rationale

> Subset schemas are derived from reference schemas. This means that a subset schema operates on the target namespace and content of the reference schema. It may act as a replacement for the reference schema, for certain application processing or human browsing.

When transforming from a reference schema to a subset schema, requirements of outside sources must be maintained. If an element is used by an outside source, then it can't be deleted. If a type uses an element, then that element must be defined. All such requirements must be kept in mind as the subset schema is constructed.

[Rule SSR5]

> The derivation of NIEM-conformant subset schemas is subject to the rules of XML Schema. No permitted transformations obviate this requirement.

Rationale

> These rules may specify that an element may be omitted, but that does not override the requirements of XSD. All types, elements, etc., that need to be validated should be included within the subset schema.

Note that using these rules to derive a schema from a valid subset schema will generate a valid subset schema. A valid subset of a valid subset is itself a valid subset.

9.1. Schema Document Element

[Rule SSR6]

> The subset schema may omit any of the following child elements of the NIEM-conformant reference schema's `xsd:schema` document element: `xsd:import, xsd:simpleType, xsd:complexType, xsd:group, xsd:attributeGroup, xsd:element, xsd:attribute`.

Rationale

> Many of the definition schema components may be omitted, if they are not otherwise required. They are "omittable." This does not mean that users must remove them, or that it won't be a violation of XSD for them to omit such components. Note that omission of an element implies omission of the element and all child elements, attributes, and namespace prefix definitions.

9.2. Annotations

[Rule SSR7]

> Any element `xsd:annotation`, `xsd:appinfo`, or `xsd:documentation` may be omitted from a subset schema.

Rationale

> Annotations are merely informative to the XSD validation process, and so may be dropped. Specific annotations may be required in reference schemas, but may be omitted from subset schemas.

9.3. Simple Type Definition

[Rule SSR8]

> An attribute `final` owned by the element `xsd:simpleType` may be expanded in scope. It may be set to `"#all"`, or to a superset of its value, or to a valid value if empty.

Rationale

> Subclasses may wish to prevent elements from being substituted via element / substitution group substitution. In such a case, the value for final may be expanded to satisfy requirements.

9.4. Simple Content Definition

Note that these rules apply to simple types, as well as to simple content in a complex type.

[Rule SSR9]

> An element `xsd:enumeration`, child of element `xsd:restriction`, may be omitted, provided that it has a sibling element `xsd:enumeration` which is not omitted. The final `xsd:enumeration` child of an element `xsd:restriction` SHALL NOT be omitted.

Rationale

> If the last xsd:enumeration is omitted, it drastically expands the set of legal values for the type.

[Rule SSR10]

> The following elements, children of element `xsd:restriction`, may be added or adjusted to reduce the set of legal values: `xsd:minExclusive, xsd:minInclusive, xsd:maxExclusive, xsd:maxInclusive, xsd:minLength, xsd:maxLength.`

[Rule SSR11]

The following elements, children of element `xsd:restriction`, may be added to reduce the set of legal values: `xsd:totalDigits, xsd:fractionDigits, xsd:length, xsd:pattern`.

Rationale

Simple type facets may be added or strengthened to limit the available set of valid values. In no case is it acceptable to enlarge the set of allowable values.

9.5. Complex Type Definition

Note that the rules specified in section 9.4, Simple Content Definition, apply to complex type definitions with simple content.

[Rule SSR12]

The attribute `block owned by` element `xsd:complexType`, or by element `xsd:element`, may be expanded in scope. It may be set to `"#all"`, or to a superset of its original value, or to a valid value if empty.

Rationale

Block prevents subtypes from being substituted for the specified element. This may be enabled or strengthened.

[Rule SSR13]

The attribute `final` owned by element `xsd:complexType` may be expanded in scope. It may be set to `"#all",` or to a superset of its value, or to a valid value if empty.

Rationale

Final prevents substitution groups from being used in element substitution. This may be enabled or strengthened

9.6. Attribute Declarations

[Rule SSR14]

The element `xsd:attribute,` when used as an attribute use schema component, may be omitted, if the value of its attribute use is "`optional`" or "`prohibited`"

Rationale

Attributes which are optional may be removed.

Note that prohibited attributes may be omitted, but that does not imply that types derived from a type with a removed prohibited attribute may add the prohibited attribute. Schemas must be built from the reference schemas, and then subset. They should not be built from the subset schemas, at the risk of invalidity or non-conformance.

[Rule SSR15]

The attribute `xsd:attribute,` when used as an attribute use schema component, MAY have a modified value which narrows the use of the attribute. If the reference value is "`optional,`" then the subset may have any value. Otherwise, it MUST have the original value.

Rationale

Optional attributes may be required or removed. Those attributes which are already required or prohibited must stay that way.

[Rule SSR16]

An element `xsd:attributeGroup` which does not act as an attribute group definition may be omitted only if all components declared by the attribute group are omittable.

Rationale

An attribute group may only be removed if all of its components are themselves removable. If any component of the attribute group is required, the attribute group must persist.

9.7. Complex Content

These rules provide methods for simplifying and reducing the model group defined in complex content.

[Rule SSR17]

An element `xsd:group, xsd:choice, or xsd:sequence`, SHALL NOT be omitted in a subset schema if its reference definition parent element is `xsd:complexType, xsd:extension,` or `xsd:restriction.`

Rationale

`group`, `choice`, and `sequence` that are roots of the particle schema component may not be eliminated, as it has substantial changes on the contents allowable by the schema construct defined by the parent element.

[Rule SSR18]

The element `xsd:group`, when used as a particle schema component, may be omitted from the subset schema, only if its reference element has a `minOccurs` attribute with a value of "`0`", or if all components declared by the group are themselves omittable.

Rationale

A group may be removed if it is, as a whole, optional.

[Rule SSR19]

The element `xsd:choice` may be omitted from the subset schema only if its reference element has a `minOccurs` attribute with a value of `"0"`, or if all components declared by the choice model group are themselves omittable.

Rationale

A choice element may be removed if it is, as a whole, optional

[Rule SSR20]

The element `xsd:sequence` may be omitted from the subset schema only if its reference element has a minOccurs attribute with a value of `"0"`, or if all components declared by the sequence model group are themselves omittable.

Rationale

A sequence may be removed if it is, as a whole, optional.

[Rule SSR21]

For any of `xsd:group` when used as a particle schema component, `xsd:element` when used as a particle schema component, `xsd:choice, or xsd:sequence,` the attributes `minOccurs` and `maxOccurs` may be adjusted to narrow the occurrence of subcomponents.

Rationale

A group may be removed if it is, as a whole, optional. Note that the "particle schema component" language is provided by **[XMLSchemaStructures]**.

[Rule SSR22]

The element `xsd:element` when used as a particle schema component may be omitted from the subset schema only if its reference element as a `minOccurs` attribute with a value of `"0"`.

Rationale

A group may be removed if it is, as a whole, optional.

9.8. Element Definition

[Rule SSR23]

The attribute `final` of element `xsd:element` may be expanded in scope. It may be set to "`#all`", or to a superset of its value, or to a valid value if empty.

Rationale

Final prevents substitution groups from being used in element substitution. This may be enabled or strengthened

[Rule SSR24]

The attribute `block` of element `xsd:element` may be expanded in scope. It may be set to "`#all`", or to a superset of its value, or to a valid value if empty.

Rationale

Block prevents subtypes from being substituted for the specified element. This may be enabled or strengthened.

[Rule SSR25]

The attribute `nillable` on an element `xsd:element` may be set to `"false"` regardless of the value of `nillable` in the reference element.

10. Constraint Schema Design Rules

[Definition GJXDOM-compatible constraint schema]

A *NIEM-compatible constraint schema* is a schema that follows the rules for NIEM-compatible constraint schemas as specified by this document.

[Rule CSR1]

A NIEM-compatible constraint schema has a `targetNamespace` identical to the `targetNamespace` of a NIEM-conformant reference schema.

Rationale

Constraint schemas operate by adding additional validation testing on already-valid content. Content must validate against NIEM-conformant reference schemas to be considered a NIEM-conformant instance.

11. Extension Schema Design Rules

12. Document Schema Design Rules

13. Conformant Instance Rules

This specification attempts to restrict XML instance data as little as possible, while still maintaining interoperability.

[Rule IND1]

> A NIEM-conformant instance MUST have a document element that is defined in a NIEM-conformant schema.

Rationale

> The root of a NIEM-conformant instance MUST be an element defined in a NIEM-conformant schema. The term *document element* is defined by **[XMLInfoSet]**.

[Rule IND2]

> A NIEM-conformant instance MUST validate to the reference schemas for namespaces contained in the instance, and for namespaces required for validation.

Rationale

> Reference schemas determine the exchange language. Derived schemas, and subsets, are for specific applications, but it is the reference schemas that set the standard for conformance.

NIEM embraces the use of XML schema instance attributes, including xsi:type, xsi:nil, and xsi:schemaLocation, as specified by **[XMLSchemaStructures]**.

[Rule IND3]

> Within a NIEM-conformant instance, the meaning of an element with no content is undefined. There SHALL NOT be a meaning assigned to an element with no content.

Rationale

> Elements without content have no specific meaning within NIEM. The lack of data should not be interpreted to mean anything other than that such data is not present.

The NIEM does not require a specific encoding, or specific requirements for the XML prolog, except as specified by **[XML]**.

14. NIEM Data Modeling Guide

This document is a developer's guide to creating XML Schema documents for use with the National Information Exchange Model (NIEM). It presents guidelines for using specific structures and idioms in NIEM-conformant XML Schema documents.

14.1. Overview of Data Modeling

This section outlines the basic techniques for creating data within NIEM, and for creating meaningful links between data items.

The paper makes a distinction between types and classes. In this section, the term "type" is used to refer to XML Schema types, which include complex types and simple types. The term "class" is used to refer to a specific entity in the data model.

A class may represent a real world object, but it may also represent any conceptual object, such as relationships and messages.

14.1.1. Properties

In order to understand how classes are created, we must understand the components that give meaning to the model: properties. A property is a component that describes a relationship between two classes. The general description is that a class *has a property*, and the *value* of the property is another class. For example, a person may have a property "person name" which has a value of "person name type".

Properties are turned into XML Schema elements for use in XML. Because of the syntax provided by XML, there are two representations of properties: *content* elements and *reference* elements. A content element is an element that, in XML, contains its value. In the following example, the element `PersonName` is a content element, because its content, in XML is an instance of its value class, "person name type".

XML example: `PersonName` is a content element

```
<Person>
  <PersonName>
    <PersonGivenName>Robert<PersonGivenName>
    <PersonSurName>Smith</PersonSurName>
  </PersonName>
  <PersonBirthDate>1970-01-01</PersonBirthDate>
</Person>
```

A reference element is a representation of a property. In a reference element, the element points to its value using a reference. A reference element indicates its value using a reference to an identifier. In the following example, `PersonNameReference` is a reference element, indicating the value of the name using a reference to the ID "A".

XML example: `PersonNameReference` is a reference element

```
<Person>
  <PersonNameReference s:reference="A"/>
  <PersonBirthDate>1970-01-01</PersonBirthDate>
</Person>

<PersonName s:id="A">
  <PersonGivenName>Robert<PersonGivenName>
  <PersonSurName>Smith</PersonSurName>
</PersonName>
```

Some properties are *containers*. A container is a property which does not establish a semantically strong relationship. The relationship described by a container property is semantically weak. A container indicates that a class (the one that has the property) has an instance of the value class. Containers generally have names based on their types; "person type" uses a container "person". The class "activity type" uses a container "activity".

For example, an "incident" may have a property "person". This indicates that an incident involved a person, but doesn't tell us what role the person played, or any additional meaning about the involvement of the person in the incident.

14.1.2. Methods for Creating Classes

There are several methods for creating data classes. Each of these methods creates new types of "things" in the data model.

14.1.2.1. Composition: Basic Class Construction

The basic method for creating classes is by composition of different parts. The parts of a class are properties. The parts composed ("put together") as a sequence of properties. These properties indicate that the class has a characteristic, a relationship, or a subpart. For example:

- A person may have the property "birth location", which indicates a relationship: the place where a person was born.
- A person may have the property "eye color", a characteristic.
- A vehicle may have the property "cargo", contents of the vehicle.

NIEM does not attempt to make concrete distinctions between these types of properties. It uses the same methods for each of them.

Properties that are put together to form a class may take the form of content elements or reference elements. Which of these two is selected is often determined by use cases and complexity. For example, a birth date would be represented as a content element. Even though lots of people could have the same birth date, and the date could be used for many purposes, it is generally easier to just use copies of the date, when it is used in multiple places.

Definition of people, however, may often take the form of a reference element. As the definition of a person may be complicated, it makes little sense to copy its value when it is needed in multiple places. It is more effective to reference a single definition, instead.

14.1.2.2. Roles

A role is a specific kind of class, which represents a particular context or activity for a thing. A role may be specific to time, incident, or employment. For example, if I pick up an object and hit someone with it, the object will take on the role of a weapon, and I will take on the role of a "justice subject", and the person I hit may take on the role of "victim". If I steal the object, the object will take the role of "stolen property".

We create a new class for a role when the role has specific data associated with it, and its own life cycle. For example, a weapon may be a role of an object, and may have a user of the weapon, an activity in which it is involved, and a description of how the weapon was used.

If there is no data specific to the role, then no new class needs to be created. In such a case, we would use the class of the thing as the value of properties, instead of creating new role classes. Take, for example, a vehicle used as a getaway car from a robbery. When designing the objects, we may take one of two options:

1. A robbery incident has a property ("getaway car") that is a "vehicle"

2. A robbery incident has a property ("getaway car") that is a role of a vehicle. The "role class" may have additional information (e.g. driver, violations, max speed, and origination point) that is specific to the vehicle's use as a getaway car.

We only need to create role classes when there is data specific to a role. We do not want to create role classes for every possible use of a particular class.

Any object may take multiple roles in a message. For example, a single person may take the role of "arresting officer", "victim", and "witness".

In XML Schema, a role is represented as a type. The type has a particular "role of" property, which indicates of what object it is a role.

XML Schema example for a weapon, a role of an object.

```
<xsd:complexType name="WeaponType">
  <xsd:sequence>
    <xsd:element ref="u:RoleOfPropertyReference" ... />
    <xsd:element ref="c:WeaponUserReference" ... />
    <xsd:element ref="c:WeaponInvolvedInActivityReference" ... />
    <xsd:element ref="c:WeaponUsageText" ... />
  </xsd:sequence>
</xsd:complexType>
```

The example shows the definition of a "weapon", which is a role of "property" (a physical object in NIEM 0.3). The element "`u:RoleOfPropertyReference`" shows which object was used as a weapon. In an instance it contains a reference to the object that was used as a weapon:

Sample XML of a weapon object

```
<Weapon>
  <u:RoleOfPropertyReference s:ref="O"/>
  <c:WeaponUserReference s:ref="P">
  <c:WeaponUsageText>Swung like a club</c:WeaponUsageText>
</Weapon>
```

This represents a weapon, which is a role of object "O", when used by person "P".

14.1.2.3. Association

An association represents a relationship between objects. It uses the methods described above. However, it is special in several ways:

1. It is labeled as an association type.
2. It represents a specific relationship between objects.
3. It contains mostly reference elements. Elements that are not reference elements should be information about the context of the relationship.

An association is used when a simple property is insufficient. Take for example, a parent-child relationship. We could represent this as simple properties:

1. The parent object has a "child" property. The value of the property is the child of the parent.
2. The child object has a "parent" property. The value of the property is the parent of the child.

These two options create concerns:

- For a given relationship, which method do we use? Do we link from the child, or from the parent, or both?
- If these are represented as content elements, what do we do about the circular reference?
- Where do we put additional information about the relationship?

To resolve these issues, we use an *association type*:

3. We create a new object that represents the relationship between the parent and the child.

An association type is composed of properties, as in the composition method. However, those properties do not describe an object. They describe a relationship. This gives us two types of properties in an association:

1. Data properties, describing the context and particulars of the relationships

2. Participants, describing the objects involved in the relationships.

For the parent-child association example, an instance may look like the following.

XML sample of a parent-child relationship

```
<NuclearFamily>
  <PersonParentReference s:ref="Person1"/>
  <PersonChildReference s:ref="Person2"/>
  <FamilyKinshipText>Adopted</FamilyKinshipText>
</NuclearFamily>
```

14.1.2.4. Specialization of Classes

Specialization is a method that creates a new class from a base class. The *base class* is some established type of thing in the data model. We create a special form of the base class called the *derived class*. We do this through *specialization*. Specialization is described by Wikipedia:

> Specialization is the opposite of generalization.
>
> Concept B is a specialization of concept A if and only if:
>
> - every instance of concept B is also an instance of concept A; and
> - there are instances of concept A which are not instances of concept B.
>
> For instance, 'Bird' is a specialization of 'Animal' because every bird is an animal, and there are animals which are not birds (dogs, for instance).

Specialization in the data model is represented in XML Schema as complex type extension. For example, a case is a special form of activity:

XML Schema sample of specialization

```
<xsd:complexType name="CaseType">
  <xsd:complexContent>
    <xsd:extension base="c:ActivityType">
      <xsd:sequence>
        <xsd:element ref="c:CaseTitleText" ... />
        <xsd:element ref="c:CaseTypeText" ... />
        <xsd:element ref="c:CaseCategoryText" ... />
        ...
        <xsd:element ref="c:CaseStatus" ... />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

In data models, specialization should only be used to create a new type of thing. It is not an appropriate way to define additional properties of the base type, as that would hinder reuse.

Specialization enables type and element substitution, where the derived class may be used where the base class is expected.

14.1.3. Additional Data Methods

There are additional methods for applying data to classes, which do not directly create new classes. Instead, these methods apply data to classes, without creating new classes.

14.1.3.1. Metadata

Metadata is a structure used to provide information about objects, in a very dynamic fashion. It is used when a certain type of data must be applied widely, without modifying existing structures.

XML instance using metadata

```
<Person>
  <PersonName s:metadata="unclassified">
    <PersonGivenName>Robert<PersonGivenName>
    <PersonSurName>Smith</PersonSurName>
  </PersonName>
  <PersonBirthDate s:metadata="classified">1970-01-01</PersonBirthDate>
</Person>

<ism:SecurityMetadata
    s:id="unclassified"
    ism:classification="U"/>
<ism:SecurityMetadata
    s:id="classified"
    ism:classification="C"
    ism:nonICmarkings="..."
    ism:releasableTo="..."
    ism:ownerProducer="..."/>
```

14.1.3.2. Augmentation

Augmentation of an object is the addition of domain- or model-specific information about a type.

NIEM is composed of numerous namespaces. These include the core NIEM namespaces (universal and common). Also included in NIEM are sanctioned domains, such as justice, immigration, and emergency management. Also working with NIEM are user-created NIEM-conformant namespaces. Each of these namespaces makes up a part of the data model for any application.

In this environment, any given part of the data model may need to add properties to existing classes. Some examples, from NIEM 0.3:

- Add a nick name to a person name (im)
- Add a distance to a relative location (im)
- Add directions to a location (em)
- Add an organizational role to a contact (em)
- Add registration information to an aircraft (intel)
- Add skillfulness information to a capability (intel)

This method stands apart from other methods, because:

1. It does not introduce new concepts. There is a need for domains to add properties to existing classes, without creating new classes for new concepts.

2. It does not define a specialized type of thing. Properties need to be added to existing classes, not specialized classes.

3. There is no relationship to represent. The new properties are not in the context of a relationship to an organization or other entity. Instead, the properties are applicable to the base object.

4. The properties are defined by a domain or other party with a focused area of interest. It is impractical to include all such properties into core or common schemas, for general use. Domains need to be able define data for their use, independent from common definitions.

5. Designers of exchanges may wish to reuse these properties; their use is not limited to a single domain.

14.2. Normalizing Element Use

All elements within types may be represented two ways:

1. A content element
2. A reference element.

14.2.1. Content Elements

Content elements enclose data. The following is an example:

```
<Person s:id="A">
  ...
  <PersonName>
    <PersonFullName>Adam Smith</PersonFullName>
  </PersonName>
  ...
</Person>
```

In this example, there is a person object. The person contains an element called `PersonName`. The `PersonName` element contains an element called `PersonFullName`. The `PersonFullName` element contains a string Adam Smith. The `PersonFullName` element is obviously a content-containing element. It has the person’s name (a literal string) as its content.

The `PersonName` is also a content-containing element, as its content represents the person name, as a structured object. It contains the element `PersonFullName`, and could contain additional elements.

14.2.2. Reference Elements

Reference elements do not enclose content. Instead, they reference content as external objects:

```
<Incident>
  <ActivityDate>2003-10-02</ActivityDate>
  ...
  <IncidentSeizedPropertyRef s:ref="C"/>
  ...
</Incident>
```

In the above example, the property that was seized as part of the incident is referenced out to another object, an XML object in the same XML instance, with the identifier `C`.

```
<Property s:id="C">
  <PropertyDescriptionText>
    White microwave oven
  </PropertyDescriptionText>
  <PropertyTypeCode>HOVEN</PropertyTypeCode>
  <PropertyMakeName>Kenmore</PropertyMakeName>
  <PropertyModelName>63292</PropertyModelName>
</Property>
```

The object that has the identifier `C` is an instance of `Property`, specifically representing a microwave oven. The reasons for representing the microwave oven outside of the incident should be quite evident: it is its own object, independent of the incident. It has its own life cycle. If the incident did not exist, the microwave oven would still exist.

The seized property is an element of the incident because it is a fixed part of the incident. The incident involved the seizing of the property, and that will not change. However, the incident should be a reference element, as the property has its own life cycle, outside of the incident.

14.2.2.1. Identifying types for reference elements

All reference elements are of the same XML Schema type: `ReferenceType` from the `structures` namespace. However, we would like to validate the XML Schema type of the thing to which the reference is referring (the *referred* object). For example:

```
<IncidentSeizedPropertyRef s:ref="C"/>
```

For `IncidentSeizedProperty`, we would like the XML Schema type of the referred object to be `PropertyType`, or something derived from that type. XML Schema does not help us here, because it does not support type checking of reference targets. XML Schema supports `XML:ID` and `XML:IDREF` types, but the constraints applied to them are few: no `ID` may be defined more than once, and any `IDREF` must refer to a defined `ID`. Beyond that, XML Schema does not help.

To define the type of referred objects, we add additional non-XSD information to the schema, which we may interpret with programs, stylesheets, or constraint languages. This additional information is added to the element definitions, and concretely specifies the type of referred objects.

```
<xs:element name="IncidentSeizedPropertyRef"
    type="s:ReferenceType">
  <xs:annotation><xs:appinfo>
    <i:referenceTarget i:name="PropertyType"/>
  </xs:appinfo></xs:annotation>
</xs:element>
```

In this example, the incident seized property is specifically defined to be of type `PropertyType` in the same namespace. Following XML Schema rules, we would expect the target of the reference to be of type `PropertyType`, or of a type properly derived from `PropertyType`.

14.2.2.2. Defining Elements

For each existing element occurring in a type:

- If the element links to a peer object, or to an independent object, then define it as a reference element
    - If the element constitutes a characteristic or subpart of the containing object, then define it as an in-line content element
    - If the element should be an association, then
        - remove it from the containing type
        - create a new association type for it
        - add the containing type as a related object
        - add the type of the original element as a related object, and
        - add properties for the association, as needed

14.3. Element Substitution

XML Schema provides numerous ways to define and use elements. Use of elements within NIEM feature two major concepts:

Types: A type represents a thing as a structured or simple value. Types represent entities or associations between entities. Types may be large and structured, with many subparts, or be simple, restricted values (e.g. a number between 1 and 10).

Elements: An element conveys the meaning, or role, of a thing. An element may be a generic, context free holder for a type (referred to as a *container*). An element may also be context-specific (referred to as a *property*). An element may have at most one type.

Elements and types are both defined by XML Schema Documents (*schemas*). Elements and types within NIEM are always defined within a namespace, the *target namespace* of the schema. Elements in schemas are defined by XML statements:

```
<xsd:element
    name="LocationCountryISO3166Alpha2Code"
    type="iso_3166:CountryAlpha2CodeType"/>
```

This defines an element with the name `LocationCountryISO3166Alpha2Code`, with the type `iso_3166:CountryAlpha2CodeType`. The element `LocationCountryISO3166Alpha2Code` is used within a type, generally within an `xsd:sequence`. This XML statement defines an element that may be used in an XML document:

```
<DocumentCoverageTextAddress>
  ...
  <LocationCountryISO3166Alpha2Code>
    US
  </LocationCountryISO3166Alpha2Code>
  ...
</DocumentCoverageTextAddress>
```

This XML data contains an element “`LocationCountryISO3166Alpha2Code`”. The content of the element (i.e. attributes along with sub-elements or simple content) is as defined by the type of the element (`LocationCountryISO3166Alpha2Code`).

14.3.1. Methods

14.3.1.1. Use `substitutionGroup`

Use `substitutionGroup` to derive elements from other elements.

The attribute `substitutionGroup` appears on element definitions. It indicates an element for which the element being defined may be substituted. Take, for example the following definitions.

In the common namespace, an element is defined that contains codes for all countries recognized by ISO 3166:

```
<xsd:element
    name="LocationCountryISO3166Alpha2Code"
    type="iso_3166:CountryAlpha2CodeType"/>
```

In a local namespace, we may define an element that contains codes for all the South American countries:

```
<xsd:element
    name="LocationSouthAmericaCountryCode"
    type="my:SouthAmericaCountryCodeType"
    substitutionGroup="c:LocationCountryISO3166Alpha2Code"/>
```

Now, in an instance, we may put `my:LocationSouthAmericaCountryCode` wherever `my:LocationCountryISO3166Alpha2Code` is expected.

1. This is an XML Schema construct; we're not creating new technology.

2. This may be done for any type of element: of complex type of simple type, of no type, extensions, derivations, etc.

3. XML Schema has very specific rules about how elements may be substituted.

4. This need not be defined all at once. Additional derivations may be created as-needed, as the NIEM model progresses.

14.3.1.2. Create root elements

Create abstract, type-less elements to represent specific concepts.

When two identical concepts are found that need separate representations, create an element as the root for the two. For example, we have two different codes for *location country*:

```
<xsd:element
    name="LocationCountryISO3166Alpha2Code"
    type="iso_3166:CountryAlpha2CodeType"/>
<xsd:element
    name="LocationCountryFIPS10-4Code"
    type="fips_10-4:CountryCodeType"/>
```

These two elements are defined independently. There is no XML Schema entity to bring them together. Any type wishing to use both of these must include both of them explicitly. We can see that we can extract a unifying concept between these two elements: "Location Country Code". We can create an element to represent this unified concept:

```
<xsd:element
    name="LocationCountryCode"
    abstract="true"/>
```

Once this element is defined, we may redefine the concrete country codes to be substitutable for this conceptual element:

```
<xsd:element
    name="LocationCountryISO3166Alpha2Code"
    type="iso_3166:CountryAlpha2CodeType"
    substitutionGroup="c:LocationCountryCode"/>
<xsd:element
    name="LocationCountryFIPS10-4Code"
    type="fips_10-4:CountryCodeType"
    substitutionGroup="c:LocationCountryCode"/>
```

The use of the `substitutionGroup` attribute brings these elements together under `LocationCountryCode`.

If we wish that a type contain *codes for country locations*, we may define it such that it includes only `c:LocationCountryCode`, and, and it will be able to carry any derived element, which have `LocationCountryCode` as their `substitutionGroup`.

There are three important characteristics of `LocationCountryCode`:

1. It is used as the `substitutionGroup` of more specific, concrete elements.

2. It has no type. This means that *any* content may be carried within a `LocationCountryCode` element. Defining the element with no type allows other elements to be substituted for it, without restriction. If the element had a type, only elements of properly derived type would be substitutable for it.

3. It is abstract. This means that a `LocationCountryCode` element is not allowed to appear within an XML document. This ensures that the `LocationCountryCode` element itself may not be used to carry content within XML instances. Since the element is untyped, it would be able to carry *arbitrary* content; having it *abstract* ensures that only well-defined data may be carried.

At this point, we have defined (1) a set of concretely-defined elements, with representations to conform to specific requirements, and (2) A few abstract base elements, from which some of the concrete elements are derived. These two steps are currently implemented in NIEM 0.2.1. In NIEM 0.2.1, the abstract elements are not used by types. Types contain the specific concrete elements, instead of the abstract conceptual elements.

14.3.1.3. Use Abstract Elements

Have types in the reference schemas contain abstract elements. Use abstract elements, when available, in type definitions within reference schemas. Doing this for `AddressType` will appear as in Listing 2 (page 7).

Keep in mind that this proposes using the abstract elements within *reference* schemas, but not necessarily within subset schemas. There are differences between the two:

Listing 1: XML Schema definition using concrete elements

```
<xsd:complexType name="AddressType">
  ...
  <xsd:sequence>
    ...
    <xsd:element ref="c:LocationCountryFIPS10-4Code" ...
    <xsd:element ref="c:LocationCountryISO3166Alpha2Code" ...
    <xsd:element ref="c:LocationCountryISO3166Alpha3Code" ...
    <xsd:element ref="c:LocationCountryISO3166NumericCode" ...
    ...
  </xsd:sequence>
  ...
</xsd:complexType>
```

Listing 2: XML Schema definition using an abstract element

```
<xsd:complexType name="AddressType">
  ...
  <xsd:sequence>
    ...
    <xsd:element ref="c:LocationCountryCode" ...
    ...
  </xsd:sequence>
  ...
</xsd:complexType>
```

1. Reference schemas are designed to be a superset of components exchanged in messages.

2. Subset schemas are created such that any XML document that validates against the subset schema will validate against the reference schema.

Subset schemas may be generated such that they *substitute* elements for the abstract elements, making them straightforward, sequenced versions of the definitions from the reference schemas.

Use of the abstract elements within type definitions in the reference schema will have the following effects:

1. Additional data types may be added for `LocationCountryCode` without modifying `AddressType`.

2. New versions of existing data types may be added, without modifying `AddressType`. For example, an update to a code may be used immediately, without waiting for an update to `AddressType`, and without type extension and type substitution methods.

3. The syntax of XML instances using element substitution is very straightforward, and generally requires less I.Q. in tools than does type substitution.

For example, here is a sample instance that uses element substitution, as proposed:

```
<DocumentCoverageTextAddress>
  ...
  <my:LocationCountryExtensionCode>
    MJQ
  </my:LocationCountryExtensionCode>
  ...
</DocumentCoverageTextAddress>
```

Here is a sample instance based on an extension of `AddressType`. Note the use of the `xsi:type` attribute.

```
<DocumentCoverageTextAddress
    xsi:type="my:ExtensionAddressType">
  ...
  <my:LocationCountryExtensionCode>
    MJQ
  </my:LocationCountryExtensionCode>
  ...
</DocumentCoverageTextAddress>
```

Reference schemas that use element substitution may be subset in a concrete manner. Subset schemas may be created that do not use element substitution. For example, the definition of `AddressType` displayed in Listing 2 (page 7) may be subset as in Listing 1 (page 7).

Tiered definitions will be easier to create. For example, a core definition of “`PersonType`” may include an abstract definition for a residence, and concrete representations may be defined in domain schemas.

This would be a refactoring process, creating abstract elements when multiple representations are needed, and using those elements in the appropriate types.

14.4. Roles

14.4.1. Description of Technique

Make the distinction between something that is a specialization of an object, and something that is a role of an object. A role is an independently valid function of an object. A role may have a life cycle independent of any specific activity. Continue to use type inheritance for specialized objects. Adopt the concept of a role as an object that represents a specific function of another object.

Define the following terms:

Base object: Some object defined in the data model

Role object: An object that represents a specific function of the base object

Base object type: The XML Schema type of the base object

Role object type: The XML Schema type of the role object

RoleOf: A property of a role object. The RoleOf property specifies the base object, of which the role object is a function.

Define schemas to account for roles:

- For each class under consideration, determine if it defines a role object.
- If it defines a role object, then:
  - Create a type to represent the class of object

- Ensure the type is *not* derived from its base object type.
- Add to the type an element `RoleOf*Reference`, referring to its base object type.

14.4.2. Syntax Examples

14.4.2.1. Instance

```
<Person s:id="P1">
  <PersonName>
    <PersonFullName>Fred Smith</PersonFullName>
  <PersonFullName>
</Person>

<EnforcementOfficial>
  <RoleOfPersonReference s:ref="P1"/>
  <EnforcementOfficialBadgeID>
    <ID>101101</ID>
  </EnforcementOfficialBadgeID>
</EnforcementOfficial>
```

Use of the element RoleOfPersonReference indicates the type of the base object (in this case, a person of type PersonType). The type is not enforced by XML Schema validation. It is indicated, and could be enforced by XSLT scripts, but is not enforced by XML Schema validation.

14.4.2.2. Type Definition

```
<complexType name="EnforcementOfficialType">
  <complexContent>
    <extension base="this:SuperType">
      <sequence>
        <element ref="this:RoleOfePersonReference"
            minOccurs="0" maxOccurs="unbounded"/>
        ...
        ... Additional elements defined for enforcement officials ...
        ...
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

14.5. Associations

Types will not be used only for representation of real-world objects. Types may also represent the association between objects. These are called *association objects*, and the types are *association types*.

14.5.1. Introduction

Data definitions within NIEM consist of types and properties. Properties represent connections between these types. These connections include:

- **Characteristics**: Values that are specific to an object, and likely invariants of that object
- **Subparts**: Objects that are smaller pieces of other objects
- **Relationships**: Connections between objects, which may be numerous and changing

Associations may be used to represent more complicated relationships than are possible with simple properties.

14.5.2. Description of Technique

14.5.2.1. Association Instance Syntax

The syntax for an instance of an association is simple. Take, for example, the marriage of Adam and Barbara Smith:

```
<MarriageAssociation>
  <SpouseRef s:ref="A"/>
  <SpouseRef s:ref="B"/>
  <MarriageDate>1937-05-12</MarriageDate>
  <DivorceDate>1973-06-02</DivorceDate>
</MarriageAssociation>
```

Interpreting the above XML fragment is straightforward:

- There is an association that we call a marriage. You can tell it is an association, and not a thing, because it is named “something association”.
- This marriage association has two spouses, a marriage date, and a divorce date.
- One spouse is referenced as the object with the identifier A. The other spouse is identified by the ID B.

These objects are specified elsewhere in the same XML instance: Object A is specified as follows:

```
<Person s:id="A">
  <PersonName>
    <PersonFullName>Adam Smith</PersonFullName>
  </PersonName>
</Person>
```

Object B is specified as follows:

```
<Person s:id="B">
  <PersonName>
    <PersonFullName>Barbara Smith</PersonFullName>
  </PersonName>
</Person>
```

Other elements in the association specify more information about the association:

```
<MarriageDate>1937-05-12</MarriageDate>
 <DivorceDate>1973-06-02</DivorceDate>
```

The marriage date and divorce date are specific to the relationship between the two spouses, and so is a natural fit for an element of the association.

14.5.2.2. Multiple Associations

An object may be involved in multiple associations, each of which is represented independently. The examples below all occur within a single XML instance, and all refer to the same object with identifier A. In this case, the object A is a person, who is an employee, a spouse, a parent, and a child.

```
<EmployerEmployeeAssociation>
  <EmployeeRef s:id="A"/>
  ...
</EmployerEmployeeAssociation>

<MarriageAssociation>
  <SpouseRef s:id="A"/>
  ...
</MarriageAssociation>

<ParentChildAssociation>
  <ParentRef s:id="A"/>
  ...
</ParentChildAssociation>

<ParentChildAssociation>
  <ChildRef s:id="A"/>
  ...
</ParentChildAssociation>
```

14.5.2.3. Schema for Associations

The definition of an association is composed of several parts:

1. An element that identifies the specific semantics of the association.
2. A type for the association. The type may be have precise semantics, or may be a more generally defined type.

14.5.2.3.1. Element definitions

For each semantically distinct association, we define an element. Each element will have annotations indicating the specific meaning of the association. Such documentation is not shown in this document, but follows the guidelines established for NIEM 3.0. The syntax is standard XML Schema. For example, here is the definition for a parent-child element:

```
<xs:element
    name="ParentChildAssociation"
    type="ParentChildAssociationType"/>
```

We may wish to define a more-specific type of parent-child association. For example, an adoptive parent-child association:

```
<xs:element
    name="AdoptiveParentChildAssociation"
    type="ParentChildAssociationType"/>
```

If we wanted to make the type specific to an adoptive parent-child situation, then we define a new type, instead of reusing the general parent-child type.

14.5.2.3.2. Association type definitions

The definition of types for associations is done as needed, depending on the content of the types. We do not, as a rule, define a new type for each use or semantic definition of an association. Instead, we define them as necessary, to accommodate the content required. Here is an example definition for a type for the parent-child association:

```
<xs:complexType name="ParentChildAssociationType">
  <xs:complexContent>
    <xs:extension base="u:AssociationType">
      <xs:sequence>
        <xs:element ref="this:ParentRef" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element ref="this:ChildRef" minOccurs="0"
            maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

The type definition has several parts:

1. The name of the type is *something* "`AssociationType`”. This makes associations between objects distinct from other types of object definitions.

2. The type is derived from another association type. This allows definition of type hierarchies for associations, and the definition of characteristics that are shared across multiple association types.

3. The content of the association is a sequence of elements. The content of the association could be entirely related objects. The association could also contain characteristics of the associations, such as dates, names, identifiers, etc.

14.5.2.4. Association type hierarchy

The use of a type hierarchy is a useful feature, but should not be overused. In the examples so far, we have seen the following:

1. A root association type, which helps group association types.

2. An association type for a parent-child association. This type had a parent and a child.

3. An association type for the marriage association.

We may wish to insert into this list of types a root type for all interpersonal associations. This, however, may be over-design, due to several factors:

1. What content would go into a generalized interpersonal association? All we would know is that the participants were people. A list of PersonRef elements is not very useful, and does not provide any semantics. The elements defined at this stage would have to be discarded to provide concrete meaning (such as spouse, parent, and child).

2. What makes an association interpersonal? Is it just that there are two people participating in the association? Would an employer-employee be an interpersonal relationship, if the employer were an individual? Would an offender-victim relationship be interpersonal? What if the victim was an organization?

3. Due to restrictions of XML Schema, we only have single-inheritance available in our toolbox; a type may have at most only a single parent. These sorts of *place-holder* types have limited usefulness, as they cannot be combined together to provide useful meaning.

Use of type inheritance should be carefully considered. Keep in mind that common types may be inserted into the type hierarchy later in model development.

14.5.3. Defining Associations

A type should be defined as an association among objects (i.e. an *AssociationType* should be created to relate the objects) only if:

1. The related objects are peers of one another and not simply a defining characteristic of or subpart of the other object(s). The term **peers** is used in a data modeling sense to mean that each object being related has its own set of characteristic property values *independently* of the other.

2. Each related object can exist independently, that is, it does not depend on the existence of the association or the other object(s). In other words, none of the objects being related should lose meaning if separated from the others.

An association may have its own characteristic attributes (properties) that either cause or result from the existence of the association. These attributes are characteristic of the association and define its nature or distinguish it from other associations and objects.

New associations should be identified based on requirements or use within IEPDs, not simply because they exist, or may be used someday.

14.6. Metadata

This technique provides a general method for applying metadata and additional content to data objects. It enables users to create a block of metadata and apply it to objects in exchanges. An object states what metadata applies to it using the `metadata` attribute.

In this example, we have a specific reported date for a person object:

```
<Person s:metadata="MD">
  <PersonName>
    <PersonGivenName>Adam</PersonGivenName>
    <PersonSurName>Brooks</PersonSurName>
  </PersonName>
  <PersonBirthDate>1960-10-07</PersonBirthDate>
</Person>

<Metadata s:id="MD">
  <ReportedDate>2005-08-01</ReportedDate>
</Metadata>
```

This example has a few interesting features:

- The person object refers to its metadata

  The reference uses the attribute `s:metadata`

  The reference is to the object with id `MD`

- The metadata is a separate element

  The element is called `Metadata`.

  The metadata object has the id `MD`.

- The ID is conveyed with the attribute `s:id`

  The metadata object contains an element `ReportedDate`

This is the core syntax. There are additional features that make this technique interesting.

14.6.1.1. Additional Cardinality of Metadata

This technique allows additional cardinality in metadata. Under GJXMD 3.0, each attribute may appear at most once. Under this method, the number of times a piece of information occurs may be controlled via the usual methods for elements in types.

```
<Metadata s:id="MD">
  <CommentText>Picked up on 12/20/02</CommentText>
  <CommentText>Released up on 12/22/02</CommentText>
  <CommentText>... additional comments ...</CommentText>
</Metadata>
```

14.6.1.2. Additional complexity of metadata

This technique allows for metadata information to be defined as is usual for elements. Elements may be of simple types, reference types, or structured types: This example has the reporting person as a reference to another person object:

```
<Metadata s:id="MD">
  <ReportingPersonRef s:ref="CD"/>
</Metadata>
```

The following example has a `ReportingPersonName` of a structured type:

```
<Metadata s:id="MD">
  <ReportingPersonName>
    <PersonGivenName>Charles</PersonGivenName>
    <PersonSurName>Davis</PersonSurName>
  </ReportingPersonName>
</Metadata>
```

14.6.1.3. Multiple blocks of metadata

This technique enables the application of multiple blocks of metadata. For example, a user may wish to apply super type metadata as well as custom metadata. The instance for this may look like the following:

```
<Person s:metadata="M1 M2">
  <PersonName>
    <PersonGivenName>Adam</PersonGivenName>
    <PersonSurName>Brooks</PersonSurName>
  </PersonName>
  <PersonBirthDate>1960-10-07</PersonBirthDate>
</Person>

<Metadata s:id="M1">
  <ReportedDate>2005-08-01</ReportedDate>
</Metadata>

<my:Metadata s:id="M2">
  <my:DatabaseID>2829019291</my:DatabaseID>
</my:Metadata>
```

This example shows two metadata blocks. Both are linked from the person object's `metadata` attribute. The first metadata block indicates the reported date. The second indicates a custom, extension database identifier.

This is made possible because of the way `s:metadata` is defined. The `metadata` attribute is of type `xsd:IDREFS`, which enables references to multiple targets. See [XMLSCHEMA-2], or [XML-INFOSET] for background on `IDREFS`.

14.6.1.4. Reuse of Metadata

This technique enables a block of metadata to be reused in multiple locations. A block of metadata may be defined once, and labeled (e.g. Source1, below). Then it may be reused by multiple objects. This creates a method similar to CSS classes, for identifying different types or sources of information.

In the following example, there are two explicit sources:

- Source 1 defines `PersonName`, `PrimaryContactInformation`, and `PersonBirthDate`
- Source 2 defines `Residence` and `Employment`.

```
<Person>
  <PersonName s:metadata="Source1">
    <PersonGivenName>Adam</PersonGivenName>
    <PersonSurName>Brooks</PersonSurName>
  </PersonName>
  <Residence s:metadata="Source2">
    ....
  </Residence>
  <PrimaryContactInformation s:metadata="Source1">
    ....
  </PrimaryContactInformation>
  <Employment s:metadata="Source2">
    ....
  </Employment>
  <PersonBirthDate s:metadata="Source1">
    1960-10-07
  </PersonBirthDate>
</Person>

<Metadata s:id="Source1">
  ... data specific to source 1 ...
</Metadata>

<Metadata s:id="Source2">
  ... data specific to source 2 ...
</Metadata>
```

14.6.1.5. Metadata Mechanism is Independent of NIEM Schema Release

This technique makes code table schemas, and additional schemas independent of the main NIEM schemas. A code schema will import the structures namespace, from which it will obtain the attribute `s:metadata`.

The following example shows a vehicle registration type code, which is of a type from NCIC.

```
<VehicleRegistration>
  <VehicleRegistrationPlateTypeCode s:metadata="PCMD">
    BU
  </VehicleRegistrationPlateTypeCode>
</VehicleRegistration>

<Metadata s:id="PCMD">
  ... metadata relevant to the plate code ...
</Metadata>
```

The schema definition for this would not need to involve a per-release proxy. Instead, all versions of the NCIC schemas could be derived from the same `structures` schema, which provides the linking mechanism. The NIEM Schema would define the element, using the type from the NCIC schema:

```
<element name="VehicleRegistrationPlateTypeCode" type="NCIC:LITType"/>
```

The NCIC schema would create a simple type for the license plate code values:

```
<xsd:simpleType name="LITSimpleType">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="AM"/>
    <xsd:enumeration value="AP"/>
    ...
    <xsd:enumeration value="VF"/>
    <xsd:enumeration value="ZZ"/>
  </xsd:restriction>
</xsd:simpleType>
```

The schema would then create a complex type. The complex type would be used as the type of elements. The type would be derived from a type in the structures namespace. This complex type would provide the `s:metadata` attribute.

```
<complexType name="LITType">
  <simpleContent>
    <restriction base="s:SimpleObjectType">
      <simpleType>
        <restriction base="this:LITSimpleType"/>
      </simpleType>
    </restriction>
  </simpleContent>
</complexType>
```

The definition of super type metadata in the main NIEM schema would indicate that it is applicable to all objects. This example applies to all *Objects* from the structures namespace.

```
<complexType name="SuperTypeMetadataType">
  <annotation><appinfo>
    <i:appliesTo i:name="Object"
        i:namespace="http://www.it.ojp.gov/structures/2.0"/>
  </appinfo></annotation>
  <complexContent>
    <extension base="s:MetadataType">
      <sequence>
        <element ref="j:CommentText"
            minOccurs="0" maxOccurs="unbounded"/>
        <element ref="j:CriminalInformationIndicator"
            minOccurs="0" maxOccurs="unbounded"/>
        ...
        <element ref="j:ReportedDate"
            minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

14.6.1.6. Metadata May be Defined to Apply to Specific Types of Objects

A metadata block may be defined to apply to a specific class of object. For example, a metadata block may apply to `PersonType` from the NIEM. This is expressed via appinfo in the schema:

```
<complexType name="MyPersonMetadataType">
<annotation><appinfo>
<i:appliesTo i:name="PersonType"
i:namespace="http://niem.gov/niem/universal/0.2"/>
</appinfo></annotation>
<complexContent>
<extension base="s:MetadataType">
<sequence>
<element ref="my:MyPersonID"
minOccurs="0" maxOccurs="unbounded"/>
...
</sequence>
</extension>
</complexContent>
</complexType>
```

Here we have a metadata block that defines the element `MyPersonID`, which may be applied to any `PersonType` object.

14.6.1.7. Metadata may be defined to apply to links between objects

A metadata block may be defined that applies to links between objects, not the objects themselves. Take as an example the special case of the name of a person:

- The person is held in custody
- The definition of the name comes from external records
- The assignment of the name to the person is based on an eyewitness

This presents three separate blocks of metadata:

- The person has one block of metadata, stating that the data was entered via booking
- The name has a block of metadata, stating when the data was validated, and the data source from which it was obtained
- The assignment of the name to the person has additional metadata, indicating the witness.

The metadata on the link is expressed with an attribute called `linkMetadata`.

```
<Person s:metatadata="PM">
  <PersonName s:metadata="PNM" s:linkMetadata="LM">
    <PersonPrefixName>Mr.</PersonPrefixName>
    <PersonGivenName>Xavier</PersonGivenName>
    <PersonMiddleName>Laughton</PersonMiddleName>
    <PersonSurName>McAlester</PersonSurName>
    <PersonSuffixName>III</PersonSuffixName>
    <PersonFullName>
      Mr. Xavier Laughton McAlester, III
    </PersonFullName>
    <PersonNameInitialsText>XLM</PersonNameInitialsText>
  </PersonName>
</Person>

<Metadata s:id="LM">
  <!-- data specific to the link between
      the person and the person name -->
  <CommentText>Reported by witness</CommentText>
  <ReportingPersonName>
    <PersonGivenName>Edward</PersonGivenName>
    <PersonSurName>Fritz</PersonSurName>
  </ReportingPersonName>
  <ReportedDate>2002-10-01</ReportedDate>
</Metadata>

<Metadata s:id="PM">
  ... data specific to the person ...
</Metadata>

<Metadata s:id="PNM">
  ... data specific to the person name ...
</Metadata>
```

The attribute `linkMetadata` conveys information that can't be conveyed by the `metadata` attribute. It tells applications that the metadata does not apply to either object. Instead, it applies to the connection between the two objects.

14.7. Class Augmentation in NIEM

14.7.1. Background

Dependence on inheritance for domain-specific extensions creates several problems. These problems include:

- Lack of reusability of domain-specific extensions.
- Difficulty of defining extensions from multiple domains.
- Overly-granular reuse of multiple-domain content: Reuse is at the element level, rather than the domain level. Types composed at that level are not interoperable.

We require a method that allows application of data to existing types, while maximizing reuse of that data and avoiding the limitations associated with an inheritance-only based extension method.

14.7.2. Terms and Concepts

In this section, the following terms are used:

- **Base type**: The type to which new data needs to be added. The base type may come from a NIEM core namespace or other NIEM-conformant namespaces.
- **Augmentation data**: Data to be added to the base type.

Augmentation of an object is the addition of domain- or model-specific information about a type. Augmentations may be provided by domains or NIEM-conformant application data models.

For example, we will need "justice-domain" data about a person. This is different than creating a new kind of person. In the real world, a person for whom justice-related data exists is not a different type of person than one that has intel-related data about them. It is most likely that a person will have both intel-related and justice-related data about them.

ZZZZZZZZZZZZZZZZZ

Error! Objects cannot be created from editing field codes.

14.7.3. Method

Use type extension to create IEPD-specific and domain model entities. Domains provide specially-defined elements as building blocks for IEPD-based extensions. These elements contain augmenting data. These elements are to be added to extensions of base types, to create derived types with augmented data.

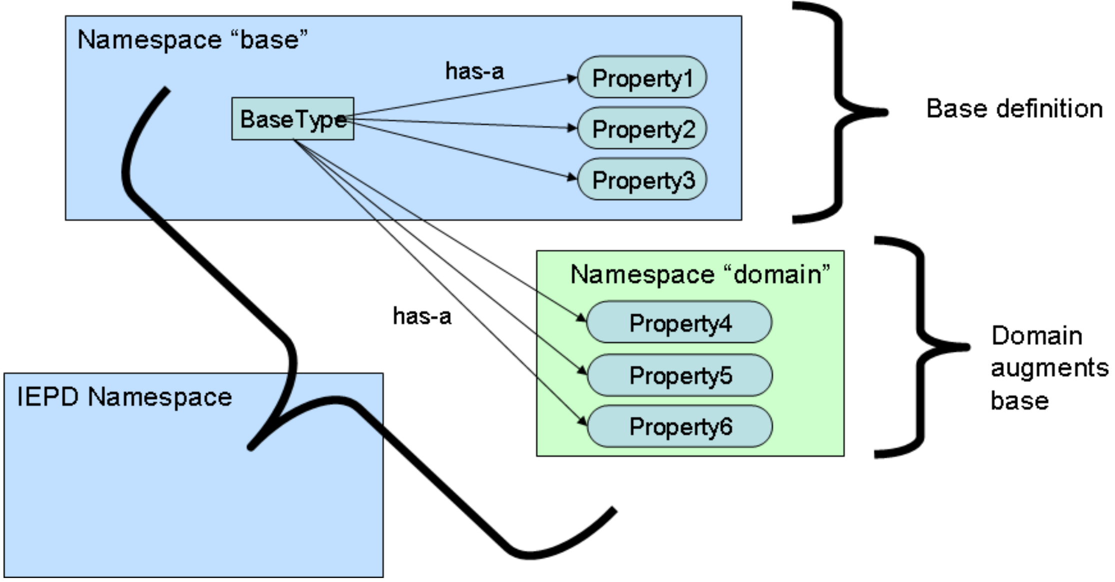
This method relies on definition of types using XML Schema type extension. This enables the use of type substitution, element substitution, and restriction.

14.7.3.1. Examples

Examples for this method use a simple set of data. There are three namespaces. Each namespace is created by a schema that is associated with NIEM. The properties used by the namespaces might be defined anywhere within NIEM. The namespaces used are:

- Namespace "`base`": This namespace models a common or core set of definitions. It contains the base object. The base object is the object to which augmentations are affixed. The base object type is named `base:BaseType`. A container for the base object is element `base:BaseContainer`. `BaseType` has properties `Property1`, `Property2`, and `Property3`.
- Namespace "`domain`": This namespace models a domain which defines augmentations to the base object. The augmentations are the elements `Property4`, `Property5`, and `Property6`. These augmentations are contained in a type named `domain:DomainAugmentationType`, when necessary.
- Namespace "`iepd`". This models an IEPD, or other concrete message definition. It contains structures and types necessary for messages to use the methods under discussion. Some methods require IEPDs to define what structures are in use. Other methods do not allow IEPDs to make such specifications, and rely on other methods for selecting augmenting data for use.

Diagram of data used for examples:



The following instance example shows an IEPD-based extension, using elements to aggregate augmenting elements. This instance looks like the IEPD extension with groups, except that the augmenting elements are wrapped in a containing element.

Sample XML instance for IEPD extension with elements:

```
<iepd:IEPDDerivedContainer>
  <base:Property1/>
  <base:Property2/>
  <base:Property3/>
  <domain:DomainAugmentationContainer>
    <domain:Property4/>
    <domain:Property5/>
    <domain:Property6/>
  </domain:DomainAugmentationContainer>
</iepd:IEPDDerivedContainer>
```

The domain schema defines `DomainAugmentationType` as an augmentation of `BaseType`. `DomainAugmentationType` is declared to be an augmentation by being an extension of `s:AugmentationType`, which is described below. The augmentation container element is declared to be an augmentation of `base:BaseType` through the use of appinfo annotations.

XML Schema for the `domain` namespace:

```
<complexType name="DomainAugmentationType">
  <complexContent>
    <extension base="s:AugmentationType">
      <sequence>
        <element ref="domain:Property4"/>
        <element ref="domain:Property5"/>
        <element ref="domain:Property6"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<element
    name="DomainAugmentationContainer"
    type="domain:DomainAugmentationType"
    substitutionGroup="s:Augmentation">
  <annotation>
    <appinfo>
      <i:appliesTo
         i:namespace="http://examples.niem.gov/ns/aug/base"
         i:name="BaseType"/>
    </appinfo>
  </annotation>
</element>
```

The proper use of such defined components may be easily verified through the use of schema verifiers.

The IEPD schema creates an extension of `BaseType`, using type extension. Note the definition of a concrete container element, substitutable for `base:BaseContainer`.

XML Schema fragment for the IEPD namespace:

```
<complexType name="IEPDDerivedType">
  <complexContent>
    <extension base="base:BaseType">
      <sequence>
        <element ref="domain:DomainAugmentationContainer"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<element
    name="IEPDDerivedContainer"
    type="iepd:IEPDDerivedType"
    substitutionGroup="base:BaseContainer"/>
```

If the IEPD used multiple augmentations, they would appear within the sequence defined by the IEPDDerivedType.

We may wish to make a rule that such augmented types must be declared as final, which would prevent them from being used as the basis for further type extension.

The domain namespace, `domain`, uses new types and elements from the structures namespace.

XML Schema fragment for the structures namespace:

```
<complexType name="AugmentationType" abstract="true">
  <attribute ref="s:metadata"/>
  <attribute ref="s:linkMetadata"/>
</complexType>

<element name="Augmentation" type="s:AugmentationType"/>
```

These structures ensure that the domain-defined types are clearly an augmentation. The use of the "Augmentation" element provides a base for element substitution, as well as tagging elements as being augmentations.

14.8. Using Non-NIEM XML Dialects with NIEM

14.8.1. Introduction

This section provides guidelines for NIEM users wishing to profile and use external standards with NIEM. In many cases employing particular standards in a NIEM Information Exchange Package Description (IEPD) may actually be preferred.

There are a variety of commonly used standards that are currently represented in XML Schema. There must be a method for NIEM to promote and use these external standards where requirements dictate.

This section focuses on a single use case: When NIEM IEPDs need to reference, import, and use components in an external standard schema or namespace that does not conform to NIEM Naming and Design Rules. It presents a methodology for including non-NIEM components in NIEM-conformant schemas. It enables data modeling efforts to build NIEM-conformant components from non-NIEM data objects.

14.8.2. Background, and Terminology

14.8.2.1. Schema Components

We use the term “schema component” for any object constructed by XML Schema. Schema components are specified by the XML Schema specification. They include attribute declarations, type definitions, etc. Some of these components may not be referenced from imported XML Schemas, and so are not concerns of this discussion. They include attribute uses (which are distinct from attribute declarations) and use of model groups (distinct from model group definitions).

From **XML Schema Part 1: Structures,** 2d Ed, W3C Recommendation, 28 October 2004:

> [Definition:]   **Schema component** is the generic term for the building blocks that comprise the abstract data model of the schema. [Definition:]   An **XML Schema** is a set of schema components. There are 13 kinds of component in all, falling into three groups. The primary components, which may (type definitions) or must (element and attribute declarations) have names are as follows:
>
> - Simple type definitions
> - Complex type definitions
> - Attribute declarations
> - Element declarations
>
> The secondary components, which must have names, are as follows:
>
> - Attribute group definitions
> - Identity-constraint definitions
> - Model group definitions
> - Notation declarations
>
> Finally, the "helper" components provide small parts of other components; they are not independent of their context:
>
> - Annotations
> - Model groups
> - Particles
> - Wildcards
> - Attribute Uses

This document is concerned only with the use of components that may be referenced from imported namespaces. Such components may be defined in one schema and used in another, when the referencing schema imports the schema that defines the component. This specification also does not pay attention to Notations and Identity constraints. Specifically, NIEM supports the referencing of the following types of components from external namespaces:

- Simple type definitions
- Complex type definitions
- Attribute declarations
- Element declarations
- Attribute group definitions
- Model group definitions

14.8.2.2. NIEM Components

We use the term “NIEM Component” for a schema component from a namespace that is NIEM-conformant, which follows the rules defined by the NIEM Naming and Design Rules (NDR) for NIEM conformance. The NIEM NDR provides a profile of W3C XML Schema, along with additional constructs to support creating a data model. In order to be NIEM-conformant, a namespace must claim conformance, and must follow specific rules about structure, XML Schema feature usage, naming, and documentation.

NIEM conformance is determined at the namespace level, based on a reference schema for a particular namespace. To determine if a namespace is NIEM-conformant, the reference schema for the namespace is tested against a set of NIEM conformance rules. These rules include such things as:

1. The schema must claim to be NIEM conformant.

2. The schema must have a target namespace, over which the schema author has dominion.

3. Schema components must be documented.

4. Component documentation must take specific forms, including being supported with XML annotations from a NIEM-specific namespace, to support data modeling concepts.

14.8.2.3. External Components

We use the term “External Component” for a schema component from a namespace that does not follow the rules for NIEM conformance.

Examples of external, non-NIEM standards include:

- GML: Geography Markup Language. GML is a prime candidate for content that may be included in NIEM structures.
- XHTML: Extensible HyperText Markup Language. This language would likely be used for exchanging simple structured text.
- SAML: Security Assertion Markup Language. This is a likely language into which NIEM content will be embedded. Some SAML assertions will likely need to contain content defined by NIEM.

14.8.3. Techniques

External components are encapsulated in NIEM-conformant components. This introduces the concept of “external adapter” types. An external adapter type is a NIEM-conformant XML Schema complex type that wraps a set of external content.

Error! Objects cannot be created from editing field codes.

These adapter types and container elements are XML Schema components, and so are defined within the namespace of the schema currently being defined.

This document specifies two constructs, which contain external content. The first is the *external adapter type*. This type is a NIEM-conformant type that contains attributes and elements from external namespaces. The second is the *external container element*. The container element is used when an external namespace provides top-level types for use, but does not provide appropriate top-level elements. In such a case, create a container element of the externally-provided type. Container elements are defined in NIEM-conformant namespaces, are named differently than regular NIEM-conformant elements, and are used in a more restricted way.

Consistent with the fundamentals of NIEM, XML elements are used for semantics, and XML Schema types are used to contain necessary structures. Specific rules for definition of adapter components will take this approach, focusing on encapsulating external structures as NIEM-conformant types, within strongly-defined elements with specific semantics.

If an external type needs to be extended for use, such extension should be done **outside** a NIEM-conformant namespace. These structures are intended to encapsulate external content. They are not indented to introduce extensions and modifications to external content into NIEM-conformant namespaces. If an application schema needs to be constructed to conform to an external standard, the schema should be created in a user-defined namespace, outside the NIEM-conformant namespaces. Then, those external components should be referenced by NIEM-conformant external adapter types and external container elements, as specified below.

14.8.4. Details

This section contains rules for using external standards in NIEM. The section uses terminology specified by **[XML-INFOSET]**. It also follows **[XMLSCHEMA-1]**.

The namespace prefix “`i`” is used in this specification as if bound to the namespace URI “`http://niem.gov/niem/appinfo/0.3`”. This namespace is used by NIEM to describe information that occurs in the schema. Such information may be used by tools to test conformance and to support the data model definition of schema content.

14.8.4.1. Namespace Conformance

A namespace can be labeled as NIEM-conformant. Any namespace that is not NIEM-conformant is referred to as an external namespace. A namespace is NIEM-conformant if its reference schema follows NIEM conformance rules. A schema component must be in a NIEM-conformant namespace to be considered NIEM conformant. For any component of a schema to be conformant, the entire schema must be conformant. A NIEM-conformant schema must claim to be conformant. This occurs when the document element, the schema element, has a child annotation with a child appinfo with a child element `i:conformant` with the character child "`true`". In other words, the XPath "`/xsd:schema/xsd:annotation/xsd:appinfo/i:conformant`" has the value "`true`".

```
<xsd:schema ...>
  <xsd:annotation>
    <xsd:appinfo>
      <i:conformant>true</i:conformant>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:schema>
```

This document only specifically addresses conformance issues for NIEM namespaces with respect to use of components from external namespaces.

14.8.4.1.1. Non-Schema Namespaces

An external namespace may be defined by a non-schema mechanism, such as DTD. In such a case, a *placeholder schema* would be created to represent the exact constructs referred to from the NIEM-conformant schema. A placeholder schema would not represent the deeper XML content of such namespaces. Instead, it would define placeholder elements and additional required constructs that are further defined by the non-XML Schema standard.

For example, XHTML 1.0, which has no normative XML Schema definition, may be considered an external namespace. XHTML defines a namespace, and numerous elements within that namespace. Were a NIEM-conformant schema specification to use the element "xhtml:ul" (an unordered list), it would use a reference. In order for schema validation to proceed normally, a schema would have to define that element. However, there is no such schema for Non-XML Schema specifications. The schema that is created to fulfill that role is the placeholder schema. Placeholder schemas should only represent the necessary components directly referred to from NIEM-conformant schemas.

14.8.4.2. Importing of External Namespaces

When NIEM namespaces are imported, the import statements are documented with a description of how the namespace is relevant to the namespace being defined. External (non-NIEM) namespaces should be documented with additional information, including:

1. An indication that the imported namespace is not NIEM-conformant.
2. The URI for a source of the reference schema for the namespace
3. Version information
4. Information about the body responsible for the standard, including:
    a. Contact information
    b. URI

Additional metadata will be defined, as the NIEM NDR is further defined. For the time being, the metadata should be included as documentation elements.

14.8.4.3. External Adapter Types

A NIEM external adapter type is a complex type that has the following qualities:

1. It is a special form of NIEM-conformant type. It may be used as the type of any NIEM-conformant element.

2. An adapter type should compose a single semantic entity. That is, the subparts of the type should appear together because they form the definition for some concept, not simply as a way of wrapping a block of external content.

3. An adapter type should be documented, as should any NIEM-conformant type.

4. It contains content from an external namespace, including:

   a. Attributes from an external namespace

   b. Attribute groups from an external namespace

   c. A single XSD sequence containing zero or more of:

      (i) Elements from an external namespace

      (ii) Model Groups from an external namespace. These are named groups of elements defined schemas.

      (iii) External container elements, from a NIEM-conformant namespace. These are used when an external *type* must be used. They are defined below.

5. It must extend the "ComplexObjectType" from the NIEM structures namespace

6. It may not directly reference any other complex or simple types. Such types should be accessed via an external container element.

7. It may not directly reference other NIEM content. Apart from the "ComplexObjectType", all content of an external adapter type should be external.

8. The content it references may be from more than one external namespace.

9. Each referenced external component must be individually documented, describing the meaning of the external component

Additional annotations may be introduced as the NDR is developed.

An example of the simple case shows an adapter type directly referring to an external element:

```
<complexType name="PointType">
  <annotation>
    <documentation>
      SUMMARY OF TYPE GOES HERE
    </documentation>
  </annotation>
  <complexContent>
    <extension base="s:ComplexObjectType">
      <sequence>
        <element ref="gml:Point">
          <annotation>
            <documentation>
              DESCRIPTION OF EXTERNAL ELEMENT GOES HERE
            </documentation>
          </annotation>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

An alternate case occurs when **types** from an external standard need to be used, instead of elements.

14.8.4.4. External Container Elements

This specification introduces the term "External" as a suffix to element names in NIEM-conformant namespaces. An element with a name that ends in "External" is referred to as an *external container element*. Such an element is defined when a NIEM standard needs to reference XML Schema types from an external namespace.

If an external namespace defines elements that are appropriate for use, the elements should be referenced by external adapter types, and external container elements are unnecessary. External container elements are needed to create container elements for types from external namespaces.

An external container element has the following characteristics:

1. Its name ends in "External".
2. It is not a NIEM-conformant element.
3. It may only be referred to by *external adapter types*. It is an error for any other component to refer to an external container element.
4. The type of the element is a simple or complex type from an external namespace. The element definition may not reference any other external components.
5. An external container element may not specify a substitution group.

External container elements may not be referenced by standard conformant components. They may only be referenced by external adapter types.

Here is an example definition of an external container element:

```
<element name="PointExternal" type="gml:PointType">
  <annotation>
    <documentation>
      DESCRIPTION OF EXTERNAL TYPE GOES HERE
    </documentation>
  </annotation>
</element>
```

Note that the definition is very simple: it provides a container for an external type, and is clearly labeled as non-NIEM content by the suffix "External".

The external container element may be used by an adapter type, as the following example shows:

```
<complexType name="PointType">
  <annotation>
    <documentation>
      SUMMARY OF TYPE GOES HERE
    </documentation>
  </annotation>
  <complexContent>
    <extension base="s:ComplexObjectType">
      <sequence>
        <element ref="this:PointExternal"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

External container elements are not NIEM-conformant data model components. Instead, they create container for external types. They are clearly identified external by their names (suffixed with "External"). External elements (that come from non-NIEM namespaces) are clearly identified as external by their namespaces.

Appendix A. Supporting Files

Appendix A.1. Schema for Structures Namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    attributeFormDefault="qualified"
    targetNamespace='http://www.it.ojp.gov/jxdm/structures/1'
    xmlns:this='http://www.it.ojp.gov/jxdm/structures/1'
    xmlns='http://www.w3.org/2001/XMLSchema'>

  <import
      namespace="http://www.w3.org/XML/1998/namespace"
      schemaLocation="xml.xsd"/>

  <attribute name="sequenceID" type="integer">

  <complexType name="ReferenceType" final="true" block="true">
    <attribute name="reference" type="IDREF" use="required"/>
    <attribute ref="xml:id" use="optional"/>
  </complexType>

  <element name="Relationship">
    <complexTypefinal="true" block="true">
      <attribute name="relationshipURI" type="anyURI"
          use="required"/>
      <attribute name="relationshipObject" type="IDREF"
          use="required"/>
      <attribute name="relationshipSubject" type="IDREF"
          use="required"/>
      <attribute ref="xml:id" use="optional"/>
    </complexType>
  </element>

</schema>
```

Appendix A.2. Schema for entity appinfo namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    targetNamespace="http://www.it.ojp.gov/jxdm/appinfo/2"
    version="1.0"
    xmlns:this="http://www.it.ojp.gov/jxdm/appinfo/2"
    xmlns="http://www.w3.org/2001/XMLSchema">

  <element name="info">
    <complexType>
      <sequence>
        <element form="qualified" maxOccurs="unbounded"
            minOccurs="0" name="base">
          <complexType>
            <attribute form="qualified" name="namespace"
                type="anyURI" use="required"/>
            <attribute form="qualified" name="name"
                type="NCName" use="required"/>
          </complexType>
        </element>
      </sequence>
      <attribute form="qualified" name="deprecated"
          type="boolean" use="required"/>
    </complexType>
  </element>

```

```
</schema>
```

Appendix A.3. Schema for xml namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    targetNamespace="http://www.w3.org/XML/1998/namespace"
    xmlns="http://www.w3.org/2001/XMLSchema">

  <attribute name="id" type="ID"/>

</schema>
```

Appendix B. Normative Abbreviations

This is a table of normative abbreviations, acronyms, and word truncations to be used as specified in .

Term	Definition
ID	Identifier
ORI	Orion value

Appendix C. References

[Global]: http://it.ojp.gov/global

[OED]: Oxford English Dictionary, Second Edition, 1989. Available at http://dictionary.oed.com/

[OJP]: OJP Information Technology Website, at http://www.it.ojp.gov/jxdm.

[RDFConcepts]: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

RDF data model: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#section-data-model

[RFC2119]: S. Bradner, Key words for use in RFCs to Indicate Requirement Levels, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.

[RFC3986]: Berners-Lee, T., et al: Uniform Resource Identifier (URI): Generic Syntax, Request for Comments 3986, January 2005. Available from http://www.ietf.org/rfc/rfc3986.txt

[SchemaForXMLSchema]: The schema for XML Schema is available at http://www.w3.org/2001/XMLSchema.xsd

[XML]: Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation 04 February 2004, available at http://www.w3.org/TR/2004/REC-xml-20040204/

EBNF notation is described at `#sec-notation`.

IDREF constraint: http://www.w3.org/TR/2004/REC-xml-20040204/#idref

[XML-ID]: xml:id Version 1.0, W3C Proposed Recommendation 12 July 2005, available from http://www.w3.org/TR/2005/PR-xml-id-20050712/.

[XMLInfoSet]: XML Information Set (Second Edition), W3C Recommendation 4 February 2004. Available from http://www.w3.org/TR/2004/REC-xml-infoset-20040204/

[XMLNamespaces]: Namespaces in XML, World Wide Web Consortium 14-January-1999, available at http://www.w3.org/TR/1999/REC-xml-names-19990114/

NCName: http://www.w3.org/TR/REC-xml-names/#NT-NCName

[XMLNamespacesErrata]: Namespaces in XML Errata, 6 December 2002, available from http://www.w3.org/XML/xml-names-19990114-errata

[XMLSchemaDatatypes]: XML Schema Part 2: Datatypes Second Edition, W3C Recommendation 28 October 2004, available at http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[XMLSchemaStructures]: XML Schema Part 1: Structures Second Edition, W3C Recommendation 28 October 2004, available at http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[XML-INFOSET]: XML Information Set (Second Edition), W3C Recommendation 4 February 2004, Available at http://www.w3.org/TR/2004/REC-xml-infoset-20040204/

IDREFS at #infoitem.attribute

[XMLSCHEMA-1]: XML Schema Part 1: Structures Second Edition, W3C Recommendation 28 October 2004. Available at http://www.w3.org/TR/2004/REC-xmlschema-1-20041028

[XMLSCHEMA-2]: XML Schema Part 2: Datatypes Second Edition, W3C Recommendation 28 October 2004. Available at http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

IDREFS at #IDREFS

Appendix D. Revision History

Revision	Date	Modifications
0.4	2005-08-23	Removed in-document tasks. Formatted for public review.
0.3	2005-08-10	Processed comments by XSTF
0.2	2005-07-21	Remove email addresses. Inserted Appendix B for acronyms.
0.1	2005-07-21	Initial draft by Webb Roberts

Appendix E. Glossary

This glossary is informative only. No definitions herein should be considered normative.

NIEM: Global Justice XML Data Model

NIEM-conformant reference schema: A schema that acts as the definition for its namespace. It maintains documentation that allows it to be shared and interoperable as a complete NIEM component.

NIEM-conformant schema: A schema that maintains the XML Schema syntax requirements of NIEM, while not necessarily containing all content for a namespace, and not necessarily containing all documentation needed for full interoperability and NIEM integration. NIEM-conformant reference schemas and NIEM-conformant subset schemas fall under this category, as do extension schemas and document schemas.

NIEM-conformant subset schema: A schema, based on a NIEM-conformant reference schema that is built to validate a subset of the content of the full reference schema. It is built from a reference schema using rules specified in this document.

NIEM constraint schema: A schema, used in conjunction with NIEM-conformant schema, that applies a set of user-designated constraints on XML data instances.

Global: The Global Justice Information Sharing Initiative. For more information, see [Global]

IEPD: Information Exchange Package Description

NIEM: National Information Exchange Model

XSTF: The Global XML Structure Task Force, the organization supervising the NIEM

Appendix F. Notices

This document and the information contained herein is provided on an “AS IS” basis and the authors DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.