XDI Core Version 1.0

Committee Specification Draft 01

29 October 2015

Specification URIs

This version:

http://docs.oasis-open.org/xdi/xdi-core/v1.0/csd01/xdi-core-v1.0-csd01.xml (Authoritative)
http://docs.oasis-open.org/xdi/xdi-core/v1.0/csd01/xdi-core-v1.0-csd01.pdf
http://docs.oasis-open.org/xdi/xdi-core/v1.0/csd01/xdi-core-v1.0-csd01.html

Previous version:

N/A

Latest version:

http://docs.oasis-open.org/xdi/xdi-core/v1.0/xdi-core-v1.0.xml (Authoritative)
http://docs.oasis-open.org/xdi/xdi-core/v1.0/xdi-core-v1.0.pdf
http://docs.oasis-open.org/xdi/xdi-core/v1.0/xdi-core-v1.0.html

Technical Committee:

OASIS XRI Data Interchange (XDI) TC

Chairs:

Drummond Reed (drummond@respect.network), Respect Network
Markus Sabadello (markus.sabadello@xdi.org), XDI.org

Editors:

Joseph Boyle (joseph@xdi.org), XDI.org
Drummond Reed (drummond@respect.network), Respect Network
Markus Sabadello (markus.sabadello@xdi.org), XDI.org

Additional artifacts:

This prose specification is one component of a Work Product which also includes:

- ABNF file: http://docs.oasis-open.org/xdi/xdi-core/v1.0/csd01/abnf/xdi-core-abnf-v1.0.txt

Abstract:

This is the core specification for XDI (Extensible Data Interchange). It defines the XDI semantic graph model, ABNF, JSON serialization, and addressing rules.

Citation format:

When referencing this specification the following citation format should be used:

[**XDI Core V1.0**]

XDI Core Version 1.0. Edited by Joseph Boyle, Drummond Reed, and Markus Sabadello. 29 October 2015. OASIS Committee Specification Draft 01. http://docs.oasis-open.org/xdi/xdi-core/v1.0/csd01/xdi-core-v1.0-csd01.html. Latest version: http://docs.oasis-open.org/xdi/xdi-core/v1.0/xdi-core-v1.0.html.

Notices

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

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see https://www.oasis-open.org/policies-guidelines/trademark for above guidance.

Table of Contents

"I would not give a fig for the simplicity this side of complexity, but I would give my life for the simplicity on the other side of complexity." —Oliver Wendell Holmes, Jr.

"Everything should be made as simple as possible, but no simpler." —Albert Einstein

"It's turtles all the way down!" —Dr. Seuss

1 Introduction

This is the core specification for XDI (Extensible Data Interchange). It defines the XDI semantic graph model, ABNF, JSON serialization, and addressing rules.

1.1 How XDI Builds On RDF

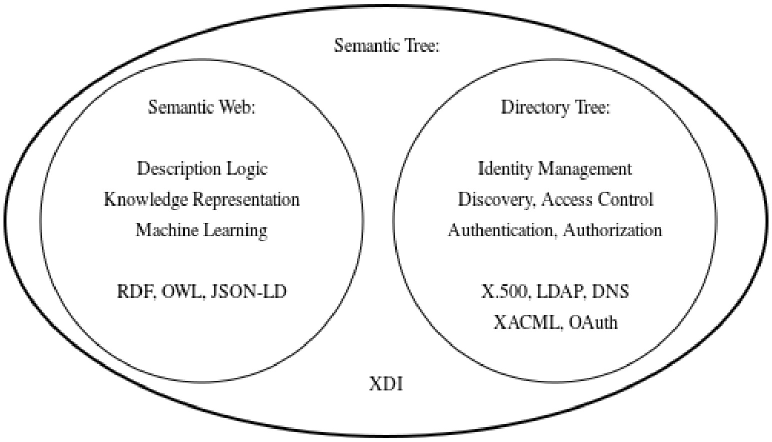
RDF (Resource Description Framework) [**rdf-concepts**] is the W3C standard for the foundation of the Semantic Web. It defines a core semantic graph model based on subject-predicate-object triples for describing data. The W3C has also defined JSON-LD [**json-ld**] and the W3C Linked Data Platform 1.0 [**ldplatform**] as W3C Recommendations.

XDI builds on the RDF graph model, adding several key features and constraints in order to optimize it for semantic data interchange. The most important of these are:

1. **Context.** XDI makes context another dimension of the graph model—to the point where contextual statements are the third fundamental type of XDI statement along with literal and relational statements.
2. **Addressability.** The XDI graph model does not allow RDF blank nodes. It also imposes the constraint that the XDI identifier of every XDI contextual arc must be unique in the scope of its parent node. The result is that every node of every XDI graph has a unique global address—and the address itself is a semantic description of that node.
3. **Simplified reification.** The XDI graph model has a standard mechanism for reification of any XDI statement, and reified statements are also uniquely addressable.
4. **Immutability.** Persistent identity is so important in distributed data sharing that XDI addressing includes special syntax for immutable identifiers of XDI graph nodes.
5. **Relativity.** Identifier scope is also critical to interoperable semantics, so XDI addressing supports explicit syntax for both absolute and relative identifiers of XDI graph nodes.
6. **Authority.** Establishing a clear chain of authority and accountability for shared data is also a key requirement of semantic data interchange. XDI contexts and classes enable directly modeling of real-world legal relationships and responsibilities.

The result is an addressable semantic tree model that brings together the benefits of Semantic Web technology with the benefits of well-established directory tree technologies, as shown in the following diagram.

Figure 1.



Additional comparisons with specific features of the RDF family of specifications will be covered throughout this specification.

1.2 Example XDI Graphs

This is an example XDI graph shown in XDI JSON serialization format. It is a relatively simple graph showing some typical profile data for a person (Alice). Alice’s graph also includes a reference to the peer graph for another person (Bob).

Note

In the examples used in this specification: a) mutable XDI identifiers (XDI names) will often use the "x-" prefix reserved for examples, and 2) immutable XDI identifiers that use XDI UUID scheme will often use truncated placeholders in the form of ":uuid:x-" for readability.

```
{
    "<$iri>": {
        "&": "https://xdi.example.com/=!:uuid:x-alice/"
    },
    "=!:uuid:x-alice": {
        "/#friend": [
            "=!:uuid:x-bob"
        ],
        "<#home><#email>": {
            "&": "alice@example.com"
        },
        "<#work><#email>": {
            "&": "asmith@example.net"
        },
        "<#personal><#email>": {
            "/$ref": [
                "=!:uuid:x-alice<#home><#email>"
            ]
        }
    },
    "=!:uuid:x-alice#passport": {
        "<#country>": {
            "&": "Canada"
        },
        "<#name>": {
            "&": "Alice Smith"
        },
        "<#num>": {
            "&": "1234567"
        }
    },
    "(=!:uuid:x-bob)": {
        "<$iri>": {
            "&": "https://xdi.example.com/=!:uuid:x-bob/"
        }
    }
}
```

This second example adds some additional XDI statements. It also adds a simple *XDI link contract*—a data sharing agreement expressed in XDI—between Alice and Bob.

```
{
    "/$is$ref": [
        "(=x-alice)",
        "(=!:uuid:x-alice)"
    ],
    "<$iri>": {
        "&": "https://xdi.example.com/=!:uuid:x-alice/"
    },
    "=!:uuid:x-alice": {
        "/#friend": [
            "=!:uuid:x-bob",
            "(=!:uuid:x-alice/#friend)"
        ],
        "/#spouse": [
            "=!:uuid:x-david"
        ],
        "[<#email>]<@~0>": {
            "&": "alice@example.com"
        },
        "[<#email>]<@~1>": {
            "&": "asmith@example.net"
        },
        "<#email>": {
            "/$ref": [
                "=!:uuid:x-alice[<#email>]<@~0>"
            ]
        },
        "<#home><#email>": {
            "/$ref": [
                "=!:uuid:x-alice[<#email>]<@~1>"
            ]
        },
        "<#work><#email>": {
            "/$ref": [
                "=!:uuid:x-alice[<#email>]<@~1>"
            ]
        }
    },
    "=!:uuid:x-alice#passport": {
        "<#country>": {
            "&": "Canada"
        },
        "<#name>": {
            "&": "Alice Smith"
        },
        "<#number>": {
            "&": "1234567"
        }
    },
    "(=!:uuid:x-alice)": {
        "/$ref": [
            ""
        ]
    },
    "(=!:uuid:x-alice/#friend)": {
        "+!:uuid:x-org#card$do": {
            "/$get": [
                "=!:uuid:x-alice<#home><#email>"
            ]
        }
    },
    "(=!:uuid:x-alice/#friend)(+!:uuid:x-org#card$do$if/$true)": {
        "{$from}": {
            "/$is#friend": [
                "=!:uuid:x-alice"
            ]
        }
    },
    "(=!:uuid:x-bob)": {
        "<$iri>": {
            "&": "https://xdi.example.com/=!:uuid:x-bob/"
        }
    }
}
```

1.3 The XDI 1.0 Specifications

XDI Core is the first of a series of specifications that will define XDI 1.0. The following table lists the other planned specifications.

Table 1. Specifications in the XDI 1.0 Suite

Specification	Description
XDI Messaging 1.0	Defines the XDI protocol as an abstract pattern for performing XDI operations using XDI messages
XDI Bindings 1.0	Defines the concrete binding of XDI messaging to specific transport protocols, beginning with the HTTP(S) protocol
XDI Link Contracts 1.0	Defines the standard structure and vocabulary of XDI authorization statements, including XDI link contracts and policy expressions, so they are portable across all XDI endpoints
XDI Discovery 1.0	Defines peer-to-peer discovery of XDI endpoint IRI(s) given an XDI address (or a discoverable identifier that can be transformed into an XDI address)
XDI Dictionary 1.0	Defines the semantic rules for XDI dictionary definitions, including a type dictionary
XDI Versioning 1.0	Defines the semantics for standardized versioning of any XDI graph or subgraph using the $v versioning context
XDI Cryptography 1.0	Defines the standards for digitally signing and encrypting XDI messages and graphs
XDI Security 1.0	Describes the mechanisms for implementing security in XDI, including transport-level security, message-level security, encryption, token formats, etc.
XDI Privacy 1.0	Describes the mechanisms for implementing privacy in XDI, including privacy-respecting identifiers (e.g., pseudonyms), privacy-respecting link contracts, data usage controls, etc.

1.4 Key Words

The key words *MUST*, *MUST NOT*, *REQUIRED*, *SHALL*, *SHALL NOT*, *SHOULD*, *SHOULD NOT*, *RECOMMENDED*, *MAY*, and *OPTIONAL* are to be interpreted as described in [**RFC 2119**].

1.5 Normative References

[**XDI-Messaging-V1.0**] *XDI 1.0 Messaging Specification*.TC work in progress. Not yet published.

[**XDI-Binding-V1.0**] *XDI 1.0 Bindings Specification*. TC work in progress. Not yet published.

[**XDI-Discovery-V1.0**] *XDI 1.0 Discovery Specification*. TC work in progress. Not yet published.

[**XDI-Dictionary-V1.0**] *XDI 1.0 Dictionary Specification*. TC work in progress. Not yet published.

[**XDI-Link-Contracts-V1.0**] *XDI 1.0 Link Contracts Specification*. TC work in progress. Not yet published.

[**XDI-Security-V1.0**] *XDI 1.0 Security Specification*. TC work in progress. Not yet published.

[**XDI-Privacy-V1.0**] *XDI 1.0 Privacy Specification*. TC work in progress. Not yet published.

[**XDI-Cryptography-V1.0**] *XDI 1.0 Cryptography Specification*. TC work in progress. Not yet published.

[**rdf-concepts**] *RDF 1.1 Concepts and Abstract Syntax* Richard Cyganiak, David Wood, Markus Lanthaler. W3C Recommendation 25 February 2014 http://www.w3.org/TR/rdf11-concepts

[**rdf-datasets**] *RDF 1.1: On Semantics of RDF Datasets* Antoine Zimmerman. W3C Working Group Note 25 February 2014 http://www.w3.org/TR/rdf11-datasets

[**rdf-schema**] *RDF 1.1 Schema* Dan Brickley, R.V. Guha. W3C Recommendation 25 February 2014 http://www.w3.org/TR/rdf-schema

[**owl**] *OWL 2 Web Ontology Language Document Overview* W3C Recommendation 11 December 2012 http://www.w3.org/TR/owl2-overview/

[**RFC 2119**] *Key words for use in RFCs to Indicate Requirement Levels* S. Bradner. BCP 14, RFC 2119, DOI 10.17487/RFC2119; IETF (Internet Engineering Task Force) March 1997 http://www.rfc-editor.org/rfc/rfc2119.txt

[**RFC 2234**] *Augmented BNF for Syntax Specifications: ABNF* D. Crocker, P. Overell. RFC 2234, DOI 10.17487/RFC2234; IETF (Internet Engineering Task Force) November 1997 http://www.rfc-editor.org/rfc/rfc2234.txt

[**RFC 3987**] *Internationalized Resource Identifiers (IRIs)* M. Duerst, M. Suignard. RFC 3987, DOI 10.17487/RFC3987; IETF (Internet Engineering Task Force) January 2005 http://www.rfc-editor.org/rfc/rfc3987.txt

[**RFC 4122**] *A Universally Unique IDentifier (UUID) URN Namespace* P. Leach M. Mealling R. Salz RFC 4122, DOI 10.17487/RFC4122; IETF (Internet Engineering Task Force) July 2005 http://www.rfc-editor.org/rfc/rfc4122.txt

[**RFC 7159**] *The JavaScript Object Notation (JSON) Data Interchange Format* T. Bray. RFC 7159, DOI 10.17487/RFC7159; IETF (Internet Engineering Task Force) March 2014 http://www.rfc-editor.org/rfc/rfc7159.txt

[**ECMA-404**] *The JSON Data Interchange Format* S. Kawamura, M. Kawashima. IETF (Internet Engineering Task Force) October 2013 http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[**UAX15**] *Unicode Normalization Forms* Mark Davis, Ken Whistler. Unicode Consortium June 2015 http://unicode.org/reports/tr15/ [http://unicode.org/reports/tr31/]

[**UAX31**] *Unicode Identifier and Pattern Syntax* Mark Davis. Unicode Consortium June 2015 http://unicode.org/reports/tr31/

1.6 Non-Normative References

[**erm**] *The Entity-Relationship Model: Toward a Unified View of Data* Peter Chen. ACM Transactions on Database Systems 1(1): 9–36. doi:10.1145/320434.320440 1976 https://en.wikiversity.org/wiki/1976/Chen

[**uml**] *UML 2.5 Specifications* ttp://www.omg.org/spec/UML/2.5/ [http://www.omg.org/spec/UML/2.5/] 1 March 2015

[**webarch**] *Architecture of the World Wide Web, Volume One* http://www.w3.org/TR/webarch/#id-resources Ian Jacobs, Norman Walsh. W3C Recommendation 15 December 2004

[**httprange14**] *HTTPRange-14* http://en.wikipedia.org/wiki/HTTPRange-14

[**reification**] *Reification (computer science)* http://en.wikipedia.org/wiki/Reification_(computer_science)

[**zooko**] *Zooko's triangle* http://en.wikipedia.org/wiki/Zooko's_triangle [**sovereign-identity**] *Sovereign Identity in the Great Silo Forest* Doc Searls Weblog http://blogs.law.harvard.edu/doc/2013/10/14/iiw-challenge-1-sovereign-identity-in-the-great-silo-forest/

[**pbd**] *Privacy by Design* https://www.privacybydesign.ca

[**json-ld**] *JSON-LD 1.0: A JSON-based Serialization for Linked Data* Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, Niklas Lindström. W3C Recommendation 16 January 2014 http://www.w3.org/TR/json-ld/

[**ldplatform**] *Linked Data Platform 1.0* Steve Speicher, John Arwe, Ashok Malhotra. W3C Recommendation 26 February 2015 http://www.w3.org/TR/ldp/

[**jsonpath**] *JSON Path* http://goessner.net/articles/JsonPath/

2 Design Goals

This section communicates the design goals that have guided the development of XDI.

2.1 100% Addressability of All Graph Nodes

To perform semantic data interchange with precise control over every data element, the first requirement of the XDI TC was that every node of every XDI graph be uniquely addressable. This architecture essentially mirrors that of the W3C in the Architecture of the World Wide Web, where it states: ""To benefit from and increase the value of the World Wide Web, agents should provide URIs as identifiers for resources.""

This requirement is one reason the XDI TC does not use the term “XDI document” or compare XDI graphs to documents. A document metaphor suggests a natural division between addressing of the document and addressing of nodes inside the document. In Web URI architecture, this is reflected by the # fragment, which represents an address local to the current resource, vs. an address outside this resource.

The XDI graph model does not have this distinction because every node of every XDI graph is equally addressable. (Or, as members of the XDI TC have put it, “It’s turtles all the way down.”)

Note that XDI addressing stops once you reach an XDI literal node—the ultimate leaf nodes of an XDI graph which contain the literal data values. If a client needs to address within a literal data value, it must switch from an XDI address to an address in the native addressing syntax of the literal data (e.g., a JSON path for a JSON document, an XML path for an XML document, a fragment for an HTML document, etc.) Such addresses are out of scope for XDI.

This requirement is perhaps the most significant difference between XDI and RDF, because unique addressability of RDF graph nodes was not part of the RDF problem domain. This is explained in more detailed in the paper *How XDI Builds on RDF* cited in the Introduction.

2.2 Heterarchical — No Central Authority

A second core design goal of XDI architecture is to support heterarchy, i.e., to not assume or rely on a central authority. This requires designing a peer-to-peer model in which any group of peers may cooperate to create an addressing and interchange space for its community. This addressing space may make use of existing resolvable identifiers for those peers, or it may extend those existing addresses, or it may be an entirely new addressing space. In all cases XDI can standardize discovery of peers and peer addresses, including both public and private discovery. This “radically P2P” architecture supports any deployment topology, from highly centralized to highly decentralized, and imposes the fewest pre-existing policy assumptions or restrictions on communities of XDI users.

Note: for more about this aspect of XDI, see the XDI Discovery specification. [**XDI-Discovery-V1.0**]

2.3 Contextual Identification

It is a mantra in digital identity that “identity is contextual”, i.e., that both the requirements for identification and the uniqueness of identifiers is relative to the context in which identification is required. Even “global” or “absolute” identifiers like telephone numbers, email addresses, or URIs are still relative to a particular addressing context.

It is also a maxim in the privacy community that “privacy is contextual”, and thus a data authority must be able to control the data being shared and permissions being granted in any identification context.

This primacy of context means that a third core XDI design goal is that it support the ability to model context at any degree of granularity and to enable XDI authorities to control the sharing of identity and data by context.

Again, we note that modeling of context was not a requirement of the RDF problem domain, so this has not been an aspect of digital identity or data sharing addressed by the RDF graph model. This topic is discussed in greater depth in the *How XDI Builds on RDF paper*.

2.4 Persistent Identification

A second core quality of identification is whether it is persistent (immutable) or reassignable (mutable). In the former case, an identifier (or other means of identification) is bound to the resource being identified in such a way that this association will not change over time—ideally forever. In the latter case, an identifier bound to one resource at one point in time (such as an IP address assigned to one computer, or a domain name registered to one owner) may subsequently be bound to a different resource at another point in time (such as when an IP address is reassigned to a new computer, or when a domain name is transferred to a new owner).

In the context of digital identity and secure data sharing, persistent identification is a requirement for one core reason: if an XDI authority with a particular identifier has been granted a specific set of permissions, and the XDI authority identified by that identifier changes, then those permissions now belong to (and can be exercised by) a different authority.

Persistent identification is also important for identity and data portability (see below), because if an identifier (or other means of identification) needs to change when the location of an XDI graph changes, the XDI relationships described in that XDI graph will break. For these reasons, it is critical that XDI syntactically distinguish a class of identifiers that XDI authorities can agree will be assigned once to a resource and never be reassigned to another resource.

At the same time, it is widely acknowledged that persistent identification is a usability nightmare. The human brain is wired to use simple, memorable natural language identifiers for our cognition and communication, and we subconsciously adjust the mappings of those identifiers over time as we learn, grow, and evolve. For example, the person you first think of by the name “Mary” today may be different from the person you first thought of by that name when you were a child.

So a key design goal of XDI is to support the requirements of both persistent and reassignable forms of identification; to provide precise means to map between them; and to make it syntactically unambiguous which form is being used in which context.

2.5 Serialization Independence

Another goal is for the XDI graph model to be a precise logical abstract model that is independent of any specified serialization format. For example, the XDI 1.0 specifications specify two display formats and one JSON serialization format. In addition the XDI TC plans to specify at least one XML serialization format. All these formats transmit 100% of the information in an XDI graph, and all must be losslessly convertible into the others.

2.6 Portability and Location Independence

Since XDI graphs may be used to describe any data associated with any entity, including people and businesses that are constantly changing contexts, attributes, service providers, and endpoints on the network, another design goal is for the semantics expressed in an XDI graph to be portable, i.e., location-independent. This means an XDI graph may be moved to any location (endpoint) on a network without breaking any of the descriptions or relations described in the graph.

This design goal is particularly important for XDI graphs representing individuals, as it supports the ability for an individual to maintain ongoing, sustainable control of his/her personal digital identity, data and relationships, independent of any particular service provider or network location.

Note: the specialized use of the XDI protocol to provide wide-area location independence is defined in the XDI Discovery specification. [**XDI-Discovery-V1.0**]

2.7 Protocol Expression and Transport Independence

To make semantic data interchange as simple and extensible as possible, another design goal is to define the XDI semantic data interchange protocol in XDI itself. This means all XDI messages are valid XDI graphs, and all XDI data sharing operations are valid XDI graph merge operations.

This design goal also achieves transport independence, i.e., as a logical protocol for the exchange of data between any two systems, the XDI protocol can be independent of any specific transport protocol (e.g., TCP/IP, HTTP(S), XMPP, SMTP, etc.), with bindings defined to such transport protocols as needed.

Note: The logical XDI protocol is defined in the XDI Messaging specification [**XDI-Messaging-V1.0**] and bindings to specific transport protocols are defined in the XDI Bindings specification. [**XDI-Binding-V1.0**]

2.8 Authorization and Policy Expression

To meet the security and privacy requirements of XDI authorities, the XDI protocol must enable them to precisely describe the rights pertaining to any shared data. Furthermore, in order for these rights to be enforced uniformly by the all XDI authorities to which they are granted, and also to be portable if an XDI graph is moved to a different location or service provider, XDI authorization must be able to be fully described in XDI itself. This includes the ability to express any policy governing authorization as well as the ability for such policies to reference data, variables, relations, and other statements in the relevant XDI graphs.

Note: the primary XDI data structure that fulfills this design goal is called a link contract and is defined in the XDI Link Contracts specification. [**XDI-Link-Contracts-V1.0**]

2.9 Schema and Ontology Expression

A key difference between markup languages and semantic data interchange is that the latter is expressly intended to solve the problem of interoperable data semantics, i.e., to provide the infrastructure necessary to map semantics between widely disparate systems with the precision necessary for digital data to automatically flow between them. To do that, it is a design goal of XDI to enable definition of schemas and ontologies for XDI data in XDI.

Note: XDI schema and ontology definition is defined in the XDI Dictionary specification. [**XDI-Dictionary-V1.0**]

2.10 Extensibility

A final design goal (and the reason for the “X” in “XDI”) is for any XDI authority to be able to extend XDI semantics without permission from any other XDI authority. This includes the ability to establish new XDI addressing spaces, to define new XDI dictionary vocabulary, and to create specializations of the XDI protocol for specific types of semantic data interchange.

3 The XDI Graph Model

To meet the design goals in the preceding section, the XDI TC developed the semantic graph model defined in this and the following sections.

3.1 Overview

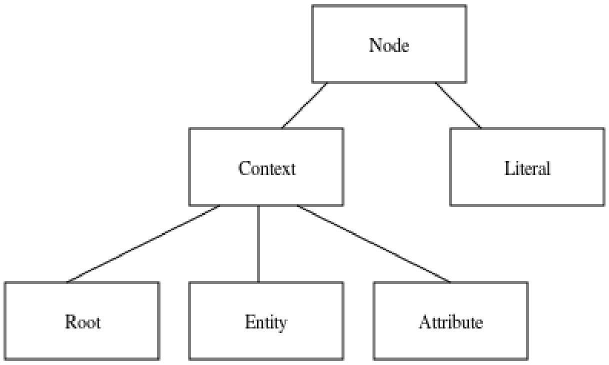
The XDI graph model builds on the RDF *subject-predicate-object* triples model [**rdf-concepts**]. This model in turn builds on the Entity-Attribute-Value (EAV) data model that dates back over 40 years. Note that in RDF, a graph node containing a data value is called a *literal*. So the RDF data model could also be termed an Entity-Attribute-Literal (EAL) model.

With RDF 1.1 datasets [**rdf-datasets**], the model was expanded to *context-subject-predicate-object* quads. The fourth component—context—represents a named RDF graph. The XDI graph model has an analogous fourth component representing the root of an XDI graph. Thus it is called the Root-Entity-Attribute-Literal (REAL) model.

3.2 Node Types

The figure below shows a simple UML class diagram (not an XDI graph) of the highest level node types in the XDI REAL graph model.

Figure 2.



All graph nodes are one of two fundamental types: *literal nodes* or *context nodes*.

3.2.1 Literal Nodes

As in RDF, XDI literal nodes are the terminal leaf nodes of the graph. They contain the raw data values described by all the other metadata in the graph. XDI natively supports the six data types defined by JSON [**RFC 7159**]:

1. Number (double-precision floating-point format in JavaScript)
2. String (double-quoted Unicode, with backslash escaping)
3. Boolean (`true` or `false`)
4. Array (an ordered, comma-separated sequence of values enclosed in square brackets; the values do not need to be of the same type)

5. Object (an unordered, comma-separated collection of key:value pairs enclosed in curly braces, with the `':'` character separating the key and the value)

6. `null` (empty—note that this is not the equivalent of *undefined*, which is when an XDI attribute has no literal node at all)

In addition to the basic data type semantics provided by JSON, the type of a literal MAY be further described using one or more XDI type statements (see *Type Relations* section).

3.2.2 Context Nodes

All non-literal nodes in the XDI graph model are called *context nodes*. In RDF the term "context" is only used to describe the top level of semantic context available in the RDF 1.1 graph model, i.e., a named RDF graph [**rdf-concepts**]. In addition, RDF blank nodes may be used to add a type of context to the relationship between other nodes. However, RDF does not use the term "context" for this purpose.

In XDI the term "context" is used uniformly across all levels of the REAL model to describe all forms of semantic context, including when:

- A graph root node provides context for another graph root node, an entity node, or an attribute node.
- An entity node provides context for another entity node or an attribute node.
- An attribute node provides context for another attribute node.

See *Contextual Arcs and Contextual Statements*, below.

All context nodes MUST have:

1. Exactly one *context type* identitied in XDI syntax by a single *context symbol*.

2. One or more *context roles* identitied in XDI syntax by zero or more pairs of *context brackets*.

3.2.3 Context Types and Symbols

The XDI REAL model defines six global context types in two groups:

1. *Classes* represent entity and attribute types.

2. *Instances* represent entity and attribute individuals.

The context symbols for each type are shown in the table below:

Table 2. Context Symbols

Group	Context type	Symbol	Words With This Symbol Also Known As
Classes	Reserved	$	keyword, dollar word
	Unreserved	#	hashtag, dictionary word
Instances	Person	=	equals name / number, person name/number
	Group	+	plus name / number, group name / number
	Thing	*	star name /number, thing name / number
	Ordinal	@	at number, order number

A definition of each context type is provided in the *Entity* section below.

Note that XDI syntax also uses four other single-character symbols:

Table 3. Other XDI Symbols

Symbol name	Symbol	Purpose	See section:
Literal symbol	&	Identify a literal arc	Literal Arcs and Statements
Immutability symbol	!	Express an immutable identifier	Mutable and Immutable Identifiers
Relativity symbol	~	Express a relative identifier	Absolute and Relative Identifiers
Triple separator	/	Separate subject, predicate, object in XDI statement formats	Statement Format

3.2.4 Context Roles and Brackets

The XDI REAL model defines six context roles in two groups:

1. *Primary roles*: every context node MUST have exactly one primary role.
2. *Secondary roles*: depending on the context, a context node MAY have one or more secondary roles.

The context brackets for each role are shown in the table below:

Table 4. Context Brackets

Group	Context role	Brackets	Also Known As
Primary	Root	( )	parentheses, parens
	Entity	none	plain, naked
	Attribute	< >	chevrons, angle brackets
Secondary	Collection	[ ]	square brackets, brackets
	Definition	| |	pipes, vertical bars
	Variable	{ }	braces, curly brackets

Each context role is defined in its own section below.

3.3 Arc Types and Statement Types

An RDF graph is a labeled directed graph in which every predicate represents a directed arc from a subject node to an object node. Each RDF subject/predicate/object statement represents exactly one such arc.

The same is true of the XDI graph model, however in XDI, an arc MUST be one of three types:

1. A *literal arc* describes the relationship between a context node and a literal node.
2. A *contextual arc* defines the identity, type, and role of one context node in the context of another context node.
3. A *relational arc* describes any other relationship between two context nodes.

Each type of arc is expressed using a specific type of XDI statement as defined in this section.

3.3.1 Literal Arcs and Literal Statements

In the XDI REAL model, a literal node MUST be the object of exactly one literal arc expressed by exactly one *literal statement*. The subject of a literal arc MUST be an XDI attribute node. An XDI attribute node MUST have no more than one literal arc.

There are two key differences between XDI literal arcs and RDF predicates whose object is a literal node:

1. *In RDF, the semantic meaning of a literal is expressed by its predicate arc.* In XDI, the semantic meaning of a literal is expressed by the sequence of XDI attribute node(s) that precede the literal arc.

2. *In RDF, a literal may have its own datatype and language attributes.* In XDI, a literal node is always an atomic leaf node. Any other semantic description of a literal node MUST be expressed using one or more XDI type statements about the parent attribute node (see *Relational Arcs and Relational Statements* section).

Because of the first difference above, an XDI literal arc is the semantic equivalent of the `rdf:value` property in RDF [**rdf-schema**]. Thus in XDI, all literal arcs MUST have the same XDI identifier: the ampersand character &. This is called the literal symbol. All XDI literal statements MUST use the literal symbol as the predicate. Examples:

Table 5. Examples of Literal Statements

Subject	Predicate	Object
`=example<#email>`	&	`"foo@example.com"`
`+example<#main><#phone>`	&	`"+44-2222-888888"`
`*!1234[<#event>]<@~78><$t>`	&	`"2010-09-20T10:11:12Z"`

In XDI JSON serialization format:

```
{
    "=example<#email>": {
        "&": "foo@example.com"
    },
    "+example<#main><#phone>": {
        "&": "+44-2222-888888"
    },
    "*!1234[<#event>]<@78><$t>": {
        "&": "2010-09-20T10:11:12Z"
    }
}
```

Because an XDI attribute node may only contain one literal node, that literal node may be uniquely addressed by appending the literal symbol & to the XDI address of the attribute node. Examples:

Table 6. Addressing a Literal Node

XDI Address of Attribute Node	XDI Address of Literal Node
`=example<#email>`	`=example<#email>&`
`+example<#main><#tel>`	`+example<#main><#tel>&`
`*!1234[<#event>]<@~78><$t>`	`*!1234[<#event>]<@~78><$t>&`

3.3.2 Contextual Arcs and Contextual Statements

In the RDF graph model, a blank node exists to provide context for other nodes. A blank node does not have a URI. It can only be identified relative to the RDF graph in which it exists. [**rdf-concepts**]

In the XDI graph model, all context nodes can provide context for other context nodes, and all context nodes are uniquely addressable. With the exception of the common root node, a context node MUST be the object of exactly one *contextual arc* expressed by exactly one contextual statement. The subject of a contextual statement MUST be another context node, called the parent node or *supercontext*. Only the common root node has no parent. The predicate of a contextual statement MUST be empty. The object of a contextual statement MUST be another context node, called the *child node* or *subcontext*. The object of a contextual statement MUST have an XDI identifier that is unique in the parent context.

The result of these requirements is that XDI context nodes form a rooted directed acyclic graph, called a *semantic tree*, in which every node is uniquely addressable and every node has a semantic meaning. The absolute XDI address of a context node is the sequence of XDI identifiers for each contextual arc that must be traversed from the common root node to the target context node.

If the common root node of an XDI graph is itself assigned a URI, all nodes in the graph become globally addressable in the universal URI addressing space as recommended by [**webarch**]. See the XDI Addressing section for details.

Following is an example of three contextual statements (each with the empty predicate) that establish the context for the final literal statement. In this example, =example and #car are XDI entities; `<#interior>` and `<#color>` are XDI attributes.

Table 7. Contextual Statements Defining A Context Path

Subject	Predicate	Object
=example		#car
=example#car		<#interior>
=example#car<#interior>		<#color>
=example#car<#interior><#color>	&	“black”

Below is the same set of statements in XDI JSON serialization format. Note that when serialized the empty predicate in a contextual statement is represented by two forward slashes:

```
{
    "=example": {
        "//": [
            "#car"
        ]
    },
    "=example#car": {
        "//": [
            "<#interior>"
        ],
        "<#interior>": {
            "//": [
                "<#color>"
            ]
        },
        "<#interior><#color>": {
            "&": "black"
        }
    }
}
```

Contextual statements are inherent in the XDI addresses of the subjects and objects of literal or relational statements. Therefore contextual statements are not included in the JSON serialization by default and are only added if they are explicitly requested using the `implied` parameter (see the Serialization section for details). Below is the same example graph without the contextual statements:

```
{
    "=example#car": {
        "<#interior><#color>": {
            "&": "black"
        }
    }
}
```

3.3.3 Relational Arcs and Relational Statements

Any relationship between two XDI graph nodes that is not described by a literal or contextual arc is described by a *relational arc* expressed by a *relational statement*. The predicate of a relational statement MUST be a sequence of one or more XDI entities.

XDI relational arcs are the equivalent of RDF properties that describe the relationship between two RDF resources. Examples:

Table 8. Examples of Relational Statements

Subject	Predicate	Object
`=person-1`	`#friend`	`=person-2`
`=person-1`	`#friend`	`=person-3`
`=person-1`	`#best#friend`	`=person-3`
`=person-1`	`#employer`	`+example.company`
`[#device]*!:uuid:...`	`#owner`	`=person-1`

In the XDI JSON serialization, a predicate expressing a relational arc is prefixed with a forward slash character:

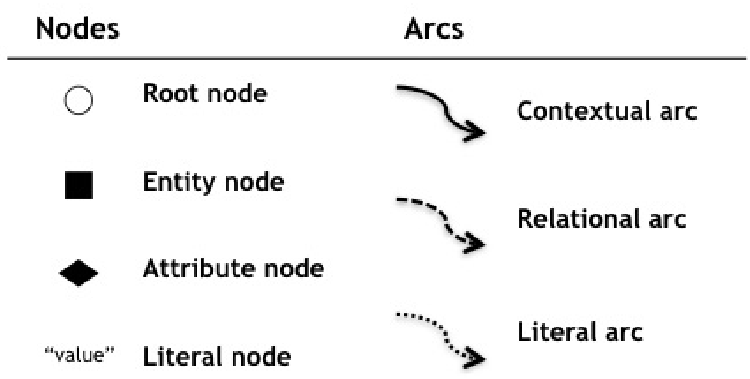
```
{
    "=person-1": {
        "/#friend": [
            "=person-2",
            "=person-3"
        ],
        "/#best#friend": [
            "=person-3"
        ],
        "/#employer": [
            "+example.company"
        ]
    },
    "[#device]*!:uuid:...": {
        "/#owner": [
            "=person-1"
        ]
    }
}
```

Relational statements may also be used to assert type or subclass relationships. See Type Relations.

3.4 Visual Graph Diagramming Notation

For consistency across implementations, the XDI Technical Committee RECOMMENDS the notation shown in the figure below for visual diagramming of XDI graphs.

Figure 3. Visual Graph Symbols



The root node symbol (a circle) is suggestive of the parentheses (   ) used in XDI syntax, and the attribute node symbol (a diamond) is suggestive of the chevron brackets <   >. The root node symbol is open to represent that an XDI graph is only a container of XDI statements. The entity and attribute node symbols are solids to represent concrete identities and properties.

For diagrams that support color, it is RECOMMENDED to use:

- A red outlined circle for the common root node.
- A blue outlined circle for a peer root node.
- A green outlined circle for an inner root node.

Literal nodes are a direct representation of the JSON value. If the value is truncated to save space, it is RECOMMENDED that the portion shown end in ellipses.

All contextual and relational arcs MUST be labeled. A literal arc MAY be labeled with the ampersand symbol, but it is not recommended. For a contextual arc, the label MUST be the unique XDI identifier of the object context node. For a relational arc, the label MUST be the predicate of the relational statement it represents.

Since there are many ways to organize an XDI graph diagram that uses this notation, the following two forms are RECOMMENDED:

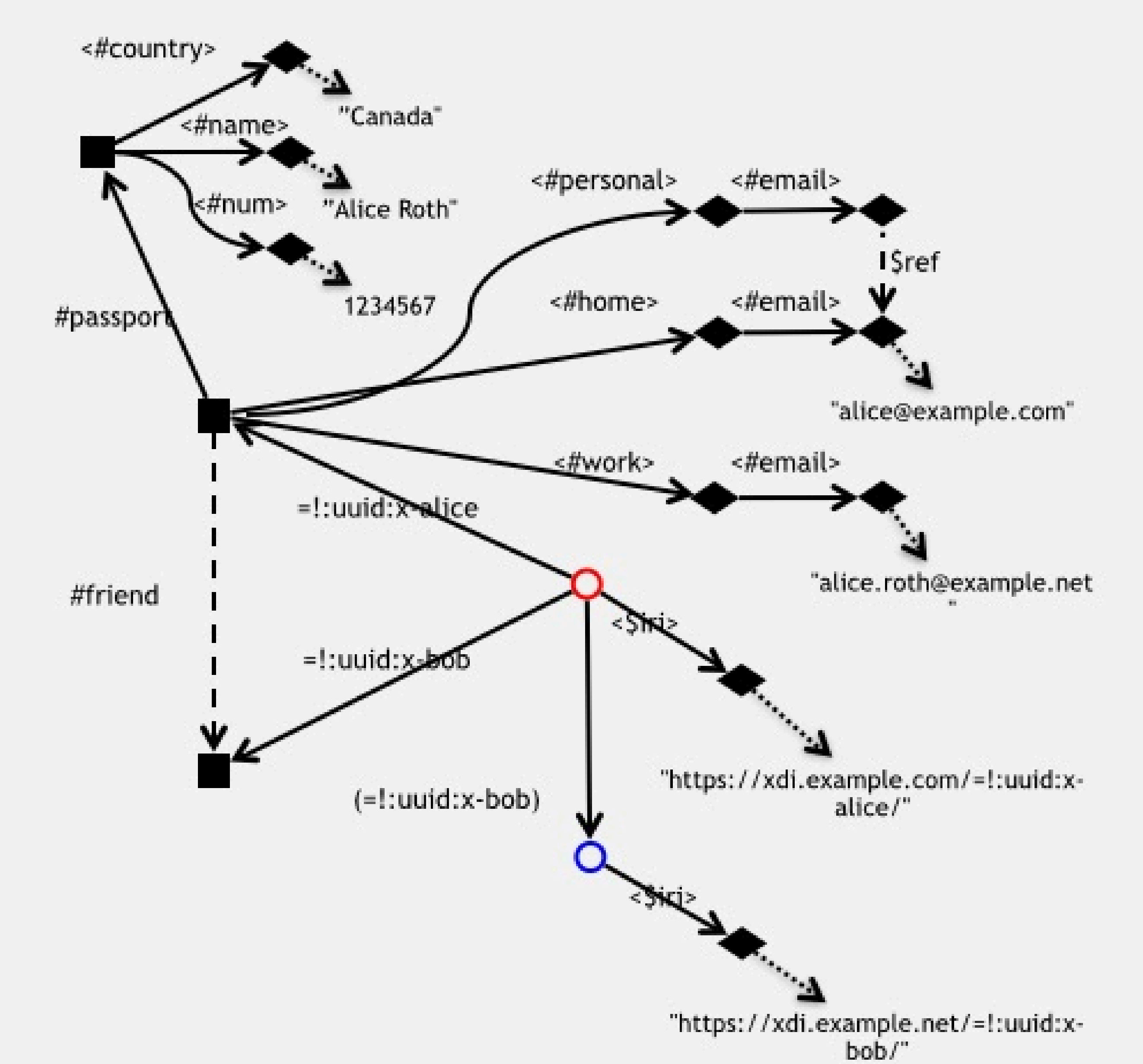
1. **Free form.** In this organization, the common root appears roughly in the center of the diagram, and arcs are arranged radiating outward from it so as to best communicate the semantic information in the graph.

2. **Tree form.** This organization mimics a typical file or directory tree layout. The common root node appears in the upper-left-hand corner, contextual and literal arcs follow a grid, and only relational arcs are curved.

The choice of form depends on the particular XDI graph being shown. It is RECOMMENDED that viewing/editing tools support both forms and enable viewers to switch between them dynamically.

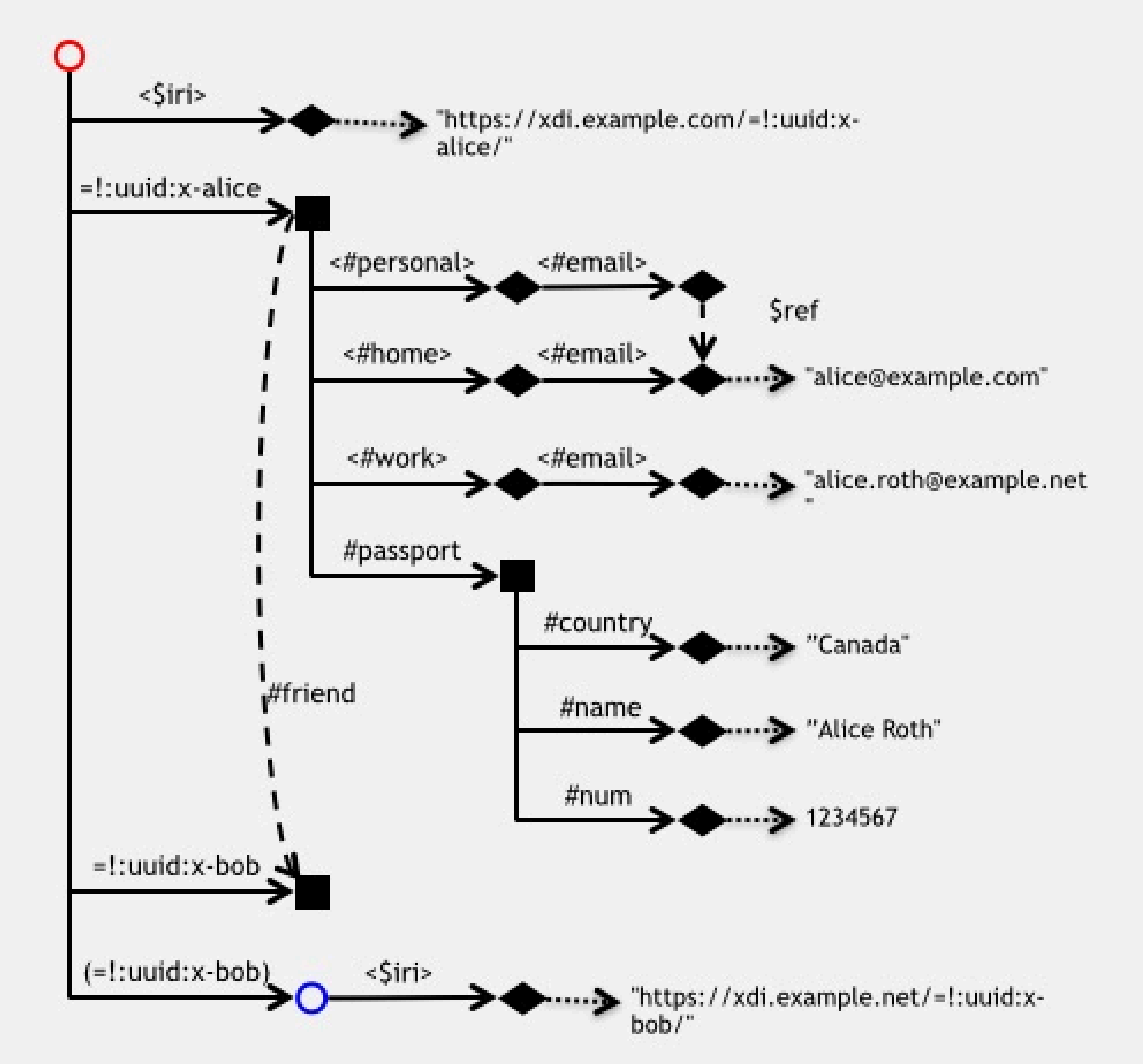
This figure shows the example XDI graph from the Introduction section in free form:

Figure 4. Graph in free form



This figure shows the same graph in tree form:

Figure 5. Graph in tree form



4 Entities

In the XDI REAL model (and the Entity-Attribute-Value model upon which it is based), an *entity* is anything (except an XDI graph itself) that can be independently identified and described, whether tangible or intangible. An entity may represent a person, group/organization, physical or digital object, concept, definition, or even a variable that may itself represent any set of these things.

From a linguistic perspective, entities are the “nouns” of XDI. However, this does not mean an entity is the only type of node that can serve as the subject of an XDI statement. In the XDI REAL model, both a root node (representing an entire XDI graph) and an attribute node may also serve as an XDI subject (and both are disjoint from entities). Thus an XDI entity is not exactly the same thing as an RDF resource—the latter may be anything with a URI (which would include XDI root and attribute nodes).

XDI entities fall into two groups: classes and instances.

4.1 Classes

A *class*, also known as a *concept* in description logic, is a set of entities that have some attribute(s) or propert(ies) in common. The set of entities belonging to the class are its *members*. In XDI, the instances of a class share the same *definition*.

XDI classes fall into two groups: reserved and unreserved.

4.1.1 Reserved ($ Symbol)

A *reserved class* is a class defined by the XDI Technical Committee to establish the universal grammar of XDI. The goal of the XDI TC is to define the smallest set of reserved classes that produce the greatest degree of semantic interoperability across XDI graphs.

The XDI identifier of a reserved class MUST begin with the $ context symbol. The $ context symbol by itself represents the class of all reserved classes. Reserved class names are also known as *dollar words* or *keywords*. Examples:

```
$iri
$do
$and
$or
$not
$public
```

A reserved class name MUST be immutable and MUST NOT use the XDI immutability symbol. A reserved class name MUST be defined in a specification from either: 1) the OASIS XDI Technical Committee (including this specification), 2) another OASIS Technical Committee specified by the OASIS XDI Technical Committee, or 3) another standards body specified by the OASIS XDI Technical Committee.

4.1.2 Unreserved (# Symbol)

An unreserved class is a class defined by any XDI authority other than the OASIS XDI Technical Committee or its specified delegate. The XDI identifier of an unreserved class MUST begin with the # context symbol. The # context symbol by itself represents the class of all unreserved classes. Unreserved class names are also known as *tags*, *hashtags*, or *dictionary words*. Examples:

```
#email
#passport
#home
```

```
#work
#friend
#enemy
```

An unreserved class name MUST be immutable and MUST NOT use the XDI immutability symbol.

In a dictionary context, unreserved class names are called *dictionary words*. Dictionary words MAY be defined by any XDI authority in any XDI context. Dictionary words whose semantics are intended to be confined to a specific set of XDI contexts SHOULD be defined in the context of the XDI authority (person or group) responsible for those contexts. Dictionary words that are intended to be generic, i.e., to share the same semantics in all XDI graphs, SHOULD be defined directly in the common root context. See Roots, below.

This begs the question of authority for generic XDI dictionary words. Like the nouns in a human language, such words represent a community consensus about shared semantics. Thus it is RECOMMENDED that generic XDI dictionary words be specified in an XDI community dictionary cooperatively maintained by the XDI authorities contributing to that community.

This is the model popularized (and proven to scale) by Wikipedia for human-readable concept definitions, and also being followed by machine-readable community ontologies such as schema.org.

4.2 Instances

An instance, also known as an *individual* in description logic, is a member of a class. The four XDI context symbols for instances are based on the fundamental nature of the context being identified:

Table 9. Instance Types

Instance type	Symbol	Personal authority?	Legal authority?	Ordered
Person	=	Yes	Yes	No
Group	+	No	Yes	No
Thing	*	No	No	No
Ordinal	@	No	No	Yes

The first three instance types are based on authority and accountability.

1. A *person* is the only entity instance that can be held personally accountable for actions taken using the XDI protocol.
2. A *group* of people (in any form) is the only entity instance that may be held legally but not personally accountable for actions taken using the XDI protocol.
3. A *thing* is an entity instance that may initiate an XDI action but cannot be held legally accountable it, such as a physical device or a software program that cannot act “of its own accord”.

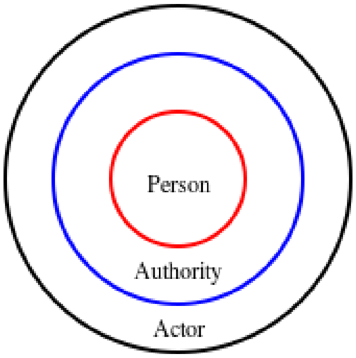
Since legal accountability plays a significant role in XDI policies and link contracts, the table below defines terms for referring to precise subsets of these three types of entity instances:

Table 10. Terminology for Instance Types

XDI term	Includes Person?	Includes Group?	Includes Thing?
XDI person	Yes	No	No
XDI authority	Yes	Yes	No
XDI actor	Yes	Yes	Yes

The legal implications of each of these terms with regard to XDI link contracts and policies is further discussed in the XDI Link Contracts 1.0 specification [**XDI-Link-Contracts-V1.0**].

Figure 6.



The final type of instance identifiers are used to specify logical order. Since XDI graphs, like RDF graphs, are unordered by default, a special class of instance identifiers called *ordinal identifiers* is needed to define explicit ordering within an XDI context.

4.2.1 Personal (= Symbol)

The XDI identifier of a natural person is called a personal authority. A personal authority MUST begin with the = context symbol (selected to suggest equality among peers). The = context symbol by itself represents the class of all personal authorities. Personal authority identifiers are also known as equal names (mutable) or equal numbers (immutable).

As explained in the XDI Addressing section, the XDI identifier for a person may be either an XDI name, an XDI number, or an encapsulated IRI. Examples of personal XDI names:

```
=example
=example-name
=example.name
```

Example of a personal XDI number (in this case, using the XDI UUID scheme):

```
=!:uuid:33ad7beb-1abc-4a26-b892-466df4379a51
```

Examples of personal encapsulated IRIs:

```
=(https://example.name/)
=(mailto:foo@example.com)
```

A personal XDI identifier represents a new form of digital identity for individuals. This can be referred to as *sovereign identity* because XDI's heterarchical and contextual graph model enables an individual to interact with other XDI authorities (both persons and groups) as an independent autonomous peer. [**sovereign-identity**].

4.2.2 Group (+ Symbol)

The second class of actor that may be held legally responsible for XDI interactions is a group — a set of people whose existence is independent of any one person. The XDI identifier of any group MUST begin with the + context symbol. The + context symbol by itself represents the class of all groups. Group identifiers are also known as *plus names* (mutable) or *plus numbers* (immutable). Examples:

```
+example
+example-company
+example.org
+!:uuid:f336a645-f5a9-41b7-ab80-ace41a8f69c2
+(https://example.com/)
+(mailto:division@example.com)
```

An XDI group identifier may represent any type of “legal person” that is not a natural person, including a group, an association, sole proprietorship, partnership, corporation, or any type of governing, political, or social body.

4.2.3 Thing (* Symbol)

Any unordered XDI instance that does not represent a person or a group represents an XDI *thing*. This includes any device, sensor, or other object that when connected to a network are commonly referred to as the Internet of Things. However it also includes any other logical “thing” such as a software program, a database, a data structure, a concept, or a unique member of a set.

Note: an XDI thing is not the same as an owl:Thing, which is the root class of all classes in OWL [**owl**].

The XDI identifier of an thing MUST begin with the * context symbol. The * context symbol by itself represents the class of all things. XDI thing identifiers are also known as *star names* (mutable) or *star numbers* (immutable).

A set of unordered instance nodes in a context MUST NOT be interpreted as having any logical order regardless of their XDI identifiers or their document order in a serialized XDI JSON document.

By itself, an XDI thing identifier does not convey any semantics about its type. The relationship of an XDI thing to a class of which it is an instance MAY be asserted in two ways:

1. **By making the thing a member of a collection.** By definition a member of a collection is an instance of the collection class. See Collections.
2. **By describing a thing instance with one or more XDI type statements.** See Type Relations.

Following is an example of XDI things serving as entity instances (in thsi case identified using immutable UUIDs):

```
{
    "+example[#item]*!:uuid:33ad7beb-1abc-4a26-b892-466df4379a51": {
        "<#price>": {
            "&": "24,995"
        }
    },
    "+example[#item]*!:uuid:f336a645-f5a9-41b7-ab80-ace41a8f69c2": {
        "<#price>": {
            "&": "36,995"
        }
    },
    "+example[#item]*!:uuid:1c958708-d5aa-4213-a6a9-73dd423502b3": {
        "<#price>": {
            "&": "18,495"
        }
    }
}
```

Following is an example of XDI things serving as attribute instances. This example also uses immutable identifiers with the XDI UUID scheme (see XDI Schemes). Note that these instance identifiers will not change even if the literal value changes.

```
{
    "=example": {
        "[<#email>]<*!:uuid:35bcc3c0-da48-df9b-a16b-0002a5d557c4>": {
            "&": "alice#example.com"
        },
        "[<#email>]<*!:uuid:fbc71e40-da47-47a6-a00e-0002a5d577b5>": {
            "&": "alice.roth@example.net"
        },
        "[<#email>]<*!:uuid:62079220-da48-21cc-aca9-0002a5d51fe6>": {
            "&": "stillalice@example.org"
        }
    }
}
```

4.2.4 Ordinal (@ Symbol)

In the absence of ordinal identifiers, the set of nodes in an XDI context or the document order of XDI statements in a serialized XDI JSON document MUST NOT be interpreted as having any logical order.

Since the order of a set of subcontexts is always relative to the parent context, to express logical order within an XDI context, the ordered subcontexts MUST use a relative ordinal identifier. A relative ordinal identifier:

1. MUST begin with the @ context symbol.

2. MUST include the ~ relativity symbol.

3. MUST be a nested identifier.

See Absolute and Relative Identifiers and Rooted and Nested Identifiers.

The @ context symbol by itself represents the class of all ordered instances.

An absolute ordinal identifier (one that does not use the ~ relativity symbol) represents the concept of a particular order position (e.g., the concept of the number "3") and not the actual relative position in an ordered sequence. Absolute ordinal identifiers MUST NOT be interpreted as being members of an ordered set.

By default, ordinal identifiers are non-negative integers, called *order numbers*. The logical order of a set of order numbers MUST begin with the number zero if present. The document order of the ordered instances in a serialized XDI JSON document MUST be ignored. An example of ordered entity instances (in this case inside a collection):

```
{
    "=example#favorite[#car]@~0": {
        "/$ref": [
            "=example[#car]*!:uuid:33ad7beb-1abc-4a26-b892-466df4379a51"

        ]
    },
    "=example#favorite[#car]@~1": {
        "/$ref": [
            "=example[#car]*!:uuid:f336a645-f5a9-41b7-ab80-ace41a8f69c2"
        ]
    },
    "=example#favorite[#car]@~2": {
        "/$ref": [
            "=example[#car]*!:uuid:1c958708-d5aa-4213-a6a9-73dd423502b3"
        ]
    }
}
```

Order numbers also apply to attribute instances as shown in this example:

```
{
    "=example": {
        "<#pref>[<#email>]<@~0>": {
            "&": "alice#example.com"
        },
        "<#pref>[<#email>]<@~1>": {
            "&": "alice.roth@example.net"
        },
        "<#pref>[<#email>]<@~2>": {
            "&": "stillalice@example.org"
        }
    }
}
```

Although order numbers are the default type of ordinal identifier, they are not the only ordering algorithm. By appending an XDI scheme, ordinal identifiers may use other ordering algorithms, such as alphabetic, alphanumeric, or byte order. See XDI Schemes.

Note: Explicit ordering of XDI graph nodes within a context using ordinal identifiers is different than canonical ordering of XDI statements for purposes of digital signatures. Canonical ordering is specified in the XDI Cryptography 1.0 specification. [**XDI-Cryptography-V1.0**]

5 Attributes < >

In the Entity-Attribute-Value (EAV) model, an attribute is a property of an entity that does not exist independently of the entity it describes. An attribute (and only an attribute) may have a literal value; an entity by itself cannot have a literal value.

In the RDF subject-predicate-object graph model, an attribute of a resource node is described by a predicate whose object is a literal node. In RDF, an attribute is not required to be unique; a resource may have multiple predicates with the same URI describing multiple literal values for the same attribute (e.g., multiple email addresses for a person).

In the XDI REAL model, all attributes are uniquely addressable because they are modeled as context nodes in an attribute role. The XDI identifier of an attribute context node MUST be an entity class or entity instance enclosed in chevron brackets < >. Examples:

```
=example<#email>
+example-company<#support><#tel>
+(https://example.com/)#shipping#address<#city>
*!: uuid:9ce739f0-7665-11e2-bcfd-0800200c18f2<#price>
```

Any type of XDI context node in any role (root, entity, attribute, collection, definition, variable) MAY have an attribute node. Note that attributes of a root node are attributes of that XDI graph as a whole and not attributes of any entity within that graph.

There are three reasons to model attributes as context nodes. First, it means all XDI attributes, like all XDI contexts, are uniquely addressable. This applies even if an entity has multiple values of the same attribute, e.g. a person with multiple email address attributes. This can be modeled as a collection where each instance is uniquely identified, as shown below.

```
{
    "=example": {
        "[<#email>]<*!:uuid:35bcc3c0-da48-df9b-a16b-0002a5d557c4>": {
            "&": "alice#example.com"
        },
        "[<#email>]<*!:uuid:fbc71e40-da47-47a6-a00e-0002a5d577b5>": {
            "&": "alice.roth@example.net"
        },
        "[<#email>]<*!:uuid:62079220-da48-21cc-aca9-0002a5d51fe6>": {
            "&": "stillalice@example.org"
        }
    }
}
```

Secondly, attributes can specialize other attributes (see *Specialization and Generalization* section). For example, `<#home>` and `<#work>` can be used to to specialize `<#email>`.

```
=example<#home><#email>
=example<#work><#email>
```

Thirdly, an attribute may itself have attributes. For example, to express the timestamp when the literal value of an attribute was assigned, add the `<$t>` (timestamp) attribute. `=example<#work><#email><$t>`

```
{
    "=example": {
        "<#work><#email>": {
            "&": "alice.roth@example.net"
        },
        "<#work><#email><$t>": {
            "&": "2010-09-20T10:11:12Z"
        }
    }
}
```

Standard XDI attributes like `$t` are defined in the XDI Dictionary specification [**XDI-Dictionary-V1.0**].

As defined in Literal Arcs and Literal Statements, only an attribute node may have a literal node, and it may have exactly zero or one literal node. The semantics of the relationship between an attribute node, its literal node, and the value of that literal node are very precise:

1. If an attribute node does not have a literal node, then the value of that attribute is *undefined*.

2. If an attribute node has a literal node, then its value is the literal JSON value of the literal node, including:

   a. `null` for a null value.

   b. `""` for an empty string.

The following example in JSON illustrates these rules.

1. `=example<#home><#email>` is undefined. Note that in this case, the attribute node must be defined with an explicit contextual statement because no statement with the literal symbol will exist.

2. `=example<#work><#email>` has the literal JSON value of null.

3. `=example<#student><#email>` has the literal JSON value of the empty string.

4. `=example<#employed>` has the literal JSON value false.

```
{
    "=example": {
        "<#home>": {
            "//": [
                "<#email>"
            ]
        },
        "<#work><#email>": {
            "&": null
        },
        "<#student><#email>": {
            "&": ""
        },
        "<#employed>": {
            "&": false
        }
    }
}
```

6 Roots ( )

After entities and attributes, the third primary role for a context node in the XDI REAL model is to represent the root of an XDI graph. While there is exactly one ultimate root node for all XDI graphs—the common root node—the heterarchical design of XDI means that it also has two other types of root nodes: peer roots and inner roots.

6.1 The Common Root

Every XDI graph MUST have exactly one common root node. It is so named because it the one logical node shared by all XDI graphs. To use the analogy of trees in a forest, if every tree represents an XDI graph, the common root node is the earth.

The XDI address of the common root node is the empty address. Thus any XDI statement that does not begin with a peer root address or an inner root address is by definition relative to the common root node. The set of all XDI context nodes that are relative only to the common root node and not to a peer root node is called the common graph.

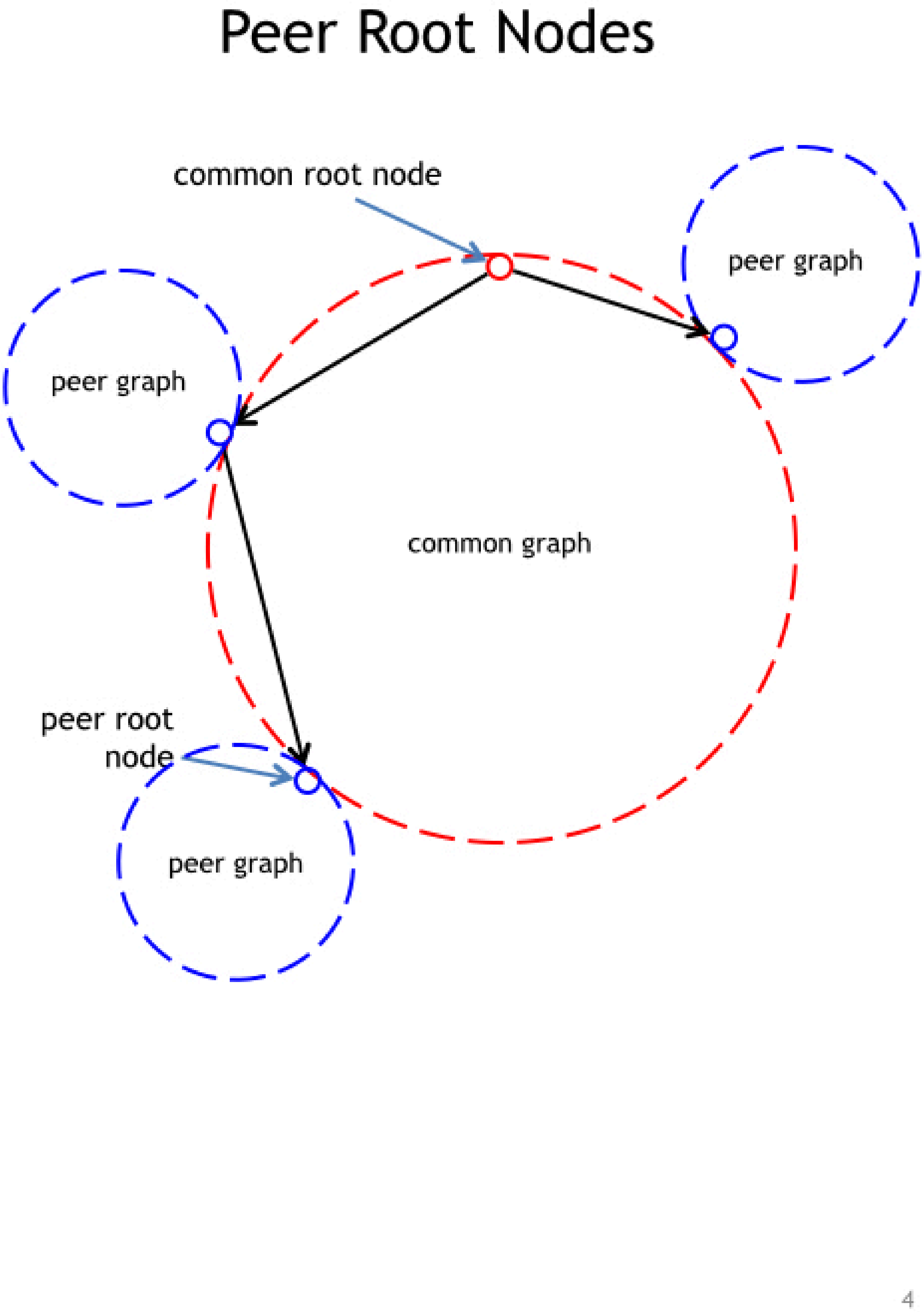
The common root node MUST NOT be the object of a direct contextual statement. It MAY be the object of an inverse contextual statement. See Inverse Relations.

The common root node of any XDI graph MAY describe the location of its own XDI endpoint using the <$iri> attribute as defined in the XDI Discovery specification [**XDI-Discovery-V1.0**]. Note that all attributes of a root node are attributes of an XDI graph as a whole and not attributes of any entity within that graph.

6.2 Peer Roots

A peer root node is a context node in one XDI graph that represents the common root node of another independent XDI graph. This concept is fundamental to XDI architecture—peer root nodes are how the XDI graph model is able to represent peer-to-peer relationships between independent XDI graphs where each graph is its own rooted tree. A node that serves as a peer root node in one XDI graph MUST serve as the common root node of its own XDI graph.

Peer root nodes may be nested to any depth within a single XDI graph. The set of contextual arcs describing these peer root nodes forms a hierarchical rooted tree. However each peer root node can be envisioned as a point on a global circle representing the logical common root node of all XDI graphs. Pick any specific starting point on this circle, and the references to the other starting points (peer roots) may be arranged hierarchically. However if you move to a different starting point, you will discover a different hierarchy. Each hierarchy represents the set of XDI peer root nodes known to a particular peer.



Peer Root Nodes

The graph contained by a peer root node is called a peer graph. A peer graph MUST be a subset of the independent XDI graph which the peer root node represents. Every peer graph is a subset of the logical XDI common graph. Thus the XDI statements in every peer graph MUST be logically consistent. (The same is not true for Inner Graphs, below).

The XDI identifier of a peer root node MUST be enclosed in parentheses ( ) and MUST NOT be preceded by an XDI context symbol. The identifier contained within the parentheses MUST be either an XDI entity identifier or an absolute URI. Examples:

```
(=example)
(+example-company)
(*!: uuid:9ce739f0-7665-11e2-bcfd-0800200c18f2)
(https://example.com/)
```

Like other context nodes, peer root nodes may be nested to any depth. This enables XDI authorities to create different XDI graphs at different XDI endpoints for different purposes and link them for the purpose of discovery. Examples:

```
(=example)(#household)
(+example-company)(#legal)(#mexico)
```

In keeping with the XDI REAL model, peer root and inner root nodes MUST precede entity or attribute nodes in the context tree. Like the common root node, a peer root node MAY use the `<$iri>` attribute to describe the network location of its XDI endpoint. Using the XDI protocol to discover the IRI for XDI peer root nodes is defined in the XDI Discovery specification [**XDI-Discovery-V1.0**]. Following is an example XDI graph from which the IRI of two peer roots can be discovered:

```
{
    "(=example)": {
        "<#iri>": {
            "&": "https://xdi.example.com/"
        }
    },
    "(=example)(#household)": {
        "<#iri>": {
            "&": "https://xdi.example.com/household/"
        }
    }
}
```

The common root node of an XDI graph may also describe its own XDI address using a direct or inverse XDI equivalence statement to a peer root node (see Equivalence Relations). This is called the graph self-description pattern and illustrated in the two example statements below.

1. Direct equivalence statement: the peer root is the subject and the common root is the object; the address of the common root is the empty string.

2. Inverse equivalence statement: the common root is the subject and the peer root is the object; the common root is represented by the outermost JSON object enclosing the entire JSON document.

```
{
    "(=example)": {
        "/$ref": [
            ""
        ]
    },
    "/$is$ref": [
        "(=example)"
    ]
}
```

Note: in the XDI Messaging specification, the term “peer” is used for an XDI agent or endpoint that sends and receives XDI messages between XDI graphs.

6.3 Inner Roots and Reification

The third type of root node plays a special role in XDI architecture. An *inner root node* represents the root of an XDI graph that is itself the object of an XDI relational statement. The graph contained by an inner root node is called an inner graph.

The XDI identifier of an inner root node MUST be enclosed in parentheses ( ) and MUST NOT be preceded by an XDI context symbol. The identifier contained within the parentheses MUST include the subject and predicate of the XDI relational statement whose object is the root node of the inner graph. The subject comes first and is separated from the predicate by a forward slash:

```
(=example/#nominated)
(+example-company/#hired)
(*!: uuid:9ce739f0-7665-11e2-bcfd-0800200c18f2/#buyer)
```

In RDF terms, each context node in an inner graph represents a reification of an XDI statement. [**reification**] The subject and predicate of the reified statement are expressed by the XDI identifier of the inner root node. The object of the statement is a context node below the inner graph. Examples:

Table 11. Reifying XDI Statements with Inner Roots

XDI Statement	XDI Address of Reified Statement
=a/#b/=c	(=a/#b)=c
=alice/#buddy/=charlie	(=alice/#buddy)=charlie

As in RDF, once a statement has been reified, it is now a new XDI statement that can serve as subject for other statements that describe the reified statement. This *inner graph pattern* is very common in XDI graphs since reification can be used to describe any relationship between two entities. Examples:

```
(=a/#b)=c/#d/=e
(=alice/#buddy)=charlie/#dentist/=edith
(=example/#hired)=abc/#employer/+example-company
(+example-company/+acquired)+other-co<$year>/&/"2014"
```

The inner graph pattern is fundamental to the structure of *link contract*s, the primary control structure in the XDI protocol. Link contracts are defined in the XDI Link Contracts specification [**XDI-Link-Contracts-V1.0**].

An inner root node or a context node within it can also serve as the object of an XDI statement:

```
=a/#b/(=c/#d)=e
```

```
=alice/#buddy/(=charlie/#dentist)=edith
```

Like other context nodes, peer root nodes MAY be nested to any depth. This enables “statements about statements about statements”. Examples:

```
(=a/#b)(=c/#d)=e/#f/=g
(=alice/#buddy)(=charlie/#dentist)=edith/#friend/=greg
```

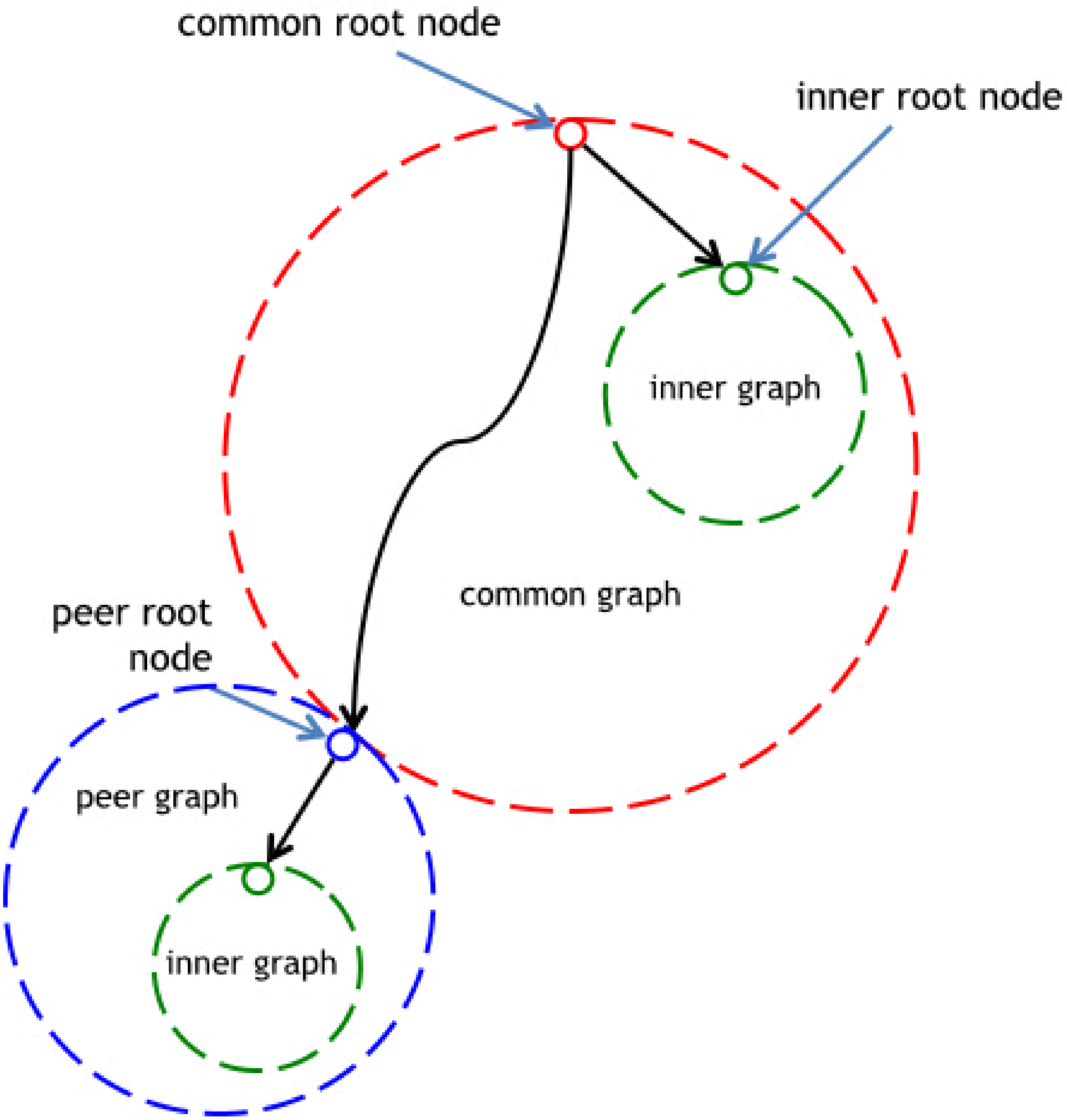
In the JSON serialization, when a sequence of peer roots and/or inner roots serves as an XDI subject, they are serialized as first level nested JSON objects. See Serialization.

```
{
    "(=alice/#buddy)": {
        "=charlie": {
            "/#dentist": [
                "=edith"
            ]
        }
    },
    "(=example/#hired)": {
        "=abc": {
            "/#employer": [
                "+example-company"
            ]
        }
    },
    "(+example-company/+acquired)": {
        "+other-co": {
            "<$year>": {
                "&": "2014"
            }
        }
    },
    "=alice": {
        "/#buddy": [
            "(=charlie/#dentist)=edith"
        ]
    }
}
```

There is an important distinction between peer graphs and inner graphs. Peer graphs are independent graphs that each contain a subset of the logical XDI common graph. By contrast, an inner graph can only be understood in the context of the unique XDI subject/predicate relationship that defines it. Therefore XDI statements in inner graphs are not required to be logically consistent with statements in the logical XDI common graph. XDI statements contained by an inner graph are relative to its specific inner root node and can only be merged with another XDI graph by also merging the containing inner root node.

Therefore inner graphs are part of the XDI common graph and can be visualized as wholly contained “graphs within graphs”.
Because peer graphs are all subsets of the logical XDI common graph, peer graphs may also contain inner graphs.

Inner Root Nodes



5

7 Collections [ ]

In the XDI REAL model, a context node whose role is to define a set of other context nodes of the same type is called a *collection*. The XDI identifier of an collection node MUST be enclosed with square brackets [ ].

A collection is either an entity collection or an attribute collection. In either case the XDI identifier of the collection node MUST be a class (i.e must start with the $ or # context symbol), called the collection class.

The set of context nodes that belong to the collection are called its members. To be a member of a collection, a context node MUST be: a) a child subcontext of the collection node, and b) a valid instance of the collection class.

A collection MAY be any class in a primary role, e.g., a root collection, an entity collection, or an attribute collection. Member instances of a collection MUST be of the same type as the collection, e.g., members of a `[#passport]` entity collection must represent instances of a passport, and members of a `[<#color>]` attribute collection must represent instances of a color.

A collection MAY contain other child nodes, however any child node that is not an ordered or unordered instance of the collection class is not a member of the collection, but a descriptor of the collection. The following example shows a collection of email address attributes with three thing instances as members, plus a timestamp attribute not a member of the collection but that describes the collection.

```
{
    "=example": {
        "[<#email>]<*!:uuid:35bcc3c0-da48-df9b-a16b-0002a5d557c4>": {
            "&": "alice@example.com"
        },
        "[<#email>]<*!:uuid:fbc71e40-da47-47a6-a00e-0002a5d577b5>": {
            "&": "alice.roth@example.net"
        },
        "[<#email>]<*!:uuid:62079220-da48-21cc-aca9-0002a5d51fe6>": {
            "&": "stillalice@example.org"
        },
        "[<#email>]<$t>": {
            "&": "2010-09-20T10:11:12Z"
        }
    }
}
```

A collection may also consist of ordered member instances, for example to indicate the priority or preference of those instances. Here is the same example showing ordered member instances:

```
{
    "=example": {
        "[<#email>]<@0>": {
            "&": "alice@example.com"
        },
        "[<#email>]<@1>": {
            "&": "alice.roth@example.net"
        },
        "[<#email>]<@2>": {
            "&": "stillalice@example.org"
        },
        "[<#email>]<$t>": {
            "&": "2010-09-20T10:11:12Z"
        }
    }
}
```

Note that in this example the order in the collection is mutable because the addresses of the ordered member instances do not include the immutability symbol. Ordering may be made immutable by adding the symbol, as shown below.

```
{
    "=example": {
        "[<#email>]<@!~0>": {
            "&": "alice@example.com"
        },
        "[<#email>]<@!~1>": {
            "&": "alice.roth@example.net"
        },
        "[<#email>]<@!~2>": {
            "&": "stillalice@example.org"
        },
        "[<#email>]<$t>": {
            "&": "2010-09-20T10:11:12Z"
        }
    }
}
```

In some cases, an XDI authority may wish to combine both mutable ordering and immutable addressing within the same collection—for example, where the order of preference is mutable but a reference to a specific unordered member of the collection will be immutable. In the XDI REAL model, the advantages of both can be combined in the same collection using $ref relations (see Equivalence Relations). This is called the ordered/unordered reference pattern. An example using email address attributes is shown below (with UUIDs shortened for readability):

```
{
    "=example": {
        "[<#email>]<@~0>": {
            "/$ref": [
                "=example[<#email>]<*!:uuid:x-1>"
            ]
        },
        "[<#email>]<@~1>": {
            "/$ref": [
                "=example[<#email>]<*!:uuid:x-2>"
            ]
        },
        "[<#email>]<@~2>": {
            "/$ref": [
                "=example[<#email>]<*!:uuid:x-3>"
            ]
        },
        "[<#email>]<*!:uuid:x-1>": {
            "&": "alice@example.com"
        },
        "[<#email>]<*!:uuid:x-2>": {
            "&": "alice.roth@example.net"
        },
        "[<#email>]<*!:uuid:x-3>": {
            "&": "stillalice@example.org"
        },
        "[<#email>]<$t>": {
            "&": "2010-09-20T10:11:12Z"
        }
    }
}
```

A context node that is not explicitly a collection is a singleton, i.e., a single instance of that context node type. Singletons and collections in XDI are analogous to singular and plural nouns in English, e.g., “passport” and “passports”. The following example shows the same three email address attribute values as in the previous example, only each one is expressed as an <#email> singleton in a different context:

```
{
    "=example": {
        "<#email>": {
            "&": "alice#example.com"
        },
        "<#home><#email>": {
            "&": "alice.roth@example.net"
        },
        "<#work><#email>": {
            "&": "stillalice@example.org"
        }
    }
}
```

By definition an attribute singleton may have only one literal value, whereas an attribute collection may contain multiple values. Again, the advantages of both can be combined in the same context using $ref relations. This is called the singleton/collection reference pattern. An example using email address attributes is shown below (with UUIDs shortened for readability):

```
{
    "=example": {
        "<#email>": {
            "/$ref": [
                "=example[<#email>]<*!:uuid:x-1>"
            ]
        },
        "<#home><#email>": {
            "/$ref": [
                "=example[<#email>]<*!:uuid:x-2>"
            ]
        },
        "<#work><#email>": {
            "/$ref": [
                "=example[<#email>]<*!:uuid:x-3>"
            ]
        },
        "[<#email>]<*!:uuid:x-1>": {
            "&": "alice@example.com"
        },
        "[<#email>]<*!:uuid:x-2>": {
            "&": "alice.roth@example.net"
        },
        "[<#email>]<*!:uuid:x-3>": {
            "&": "stillalice@example.org"
        },
        "[<#email>]<$t>": {
            "&": "2010-09-20T10:11:12Z"
        }
    }
}
```

8 Definitions | |

XML has schemas; RDF has ontologies; XDI has *dictionaries*. XDI uses the term “dictionary” for XDI ontology definitions because an XDI ontology term may be reused in many different XDI contexts just like a natural language word may be reused in many different linguistic contexts.

As with a natural language dictionary, a subject node in an XDI dictionary is called a *definition*. Each XDI definition is the subject of one or more definition statements. The XDI identifier of a definition context node MUST be enclosed with pipe symbols | |.

Definitions only apply to XDI entities or attributes. Certain definition statement types apply to each. For example, only an attribute definition may define the datatype of a literal value.

The standard attributes and relations for XDI definition statements are defined in the XDI Dictionary 1.0 specification. These include the same basic ontological building blocks as in RDFS [**rdf-schema**] and OWL [**owl**], e.g.:

- types
- subtypes
- supertypes
- entities (for roots)
- subentities (for entities)
- superentities (for entities and attributes)
- attributes (for roots and entities)
- subattributes (for attributes)
- superattributes (for attributes)
- datatypes (for literals)
- incoming relations (range)
- outgoing relations (domain)
- cardinality

Following is an example generic dictionary definition illustrating a number of common dictionary statement types (in order: entity type, outgoing relations, subentities and attributes, cardinality, and attribute types).

```
{
    "|#car|": {
        "/$is#": [
            "|#vehicle|"
        ],
        "(/)": [
            "|#owner|",
            "|#driver|",
            "|#insurer|"
        ],
        "//": [
            "|#engine|",
            "|#door|",
            "|<#model>|",
            "|<$year>|"
        ]
    },
    "|#car||#engine|": {
        "<$n>": {
            "&": "1"
        }
    },
    "|#car||#door|": {
        "<$n>": {
            "&": "2-4"
        }
    },
    "|#car||<#model>|": {
        "$is#": [
            "|<$string>|"
        ]
    },
    "|#car||<#year>|": {
        "$is#": [
            "|<$number>|"
        ]
    }
}
```

Using contextual statements, XDI dictionaries can also define superentities and superattributes. Following is an example of defining superentities for a `#car`:

```
{
    "|#car|": {
        "/$is()": [
            "|#sports|",
            "|#race|",
            "|#economy|",
            "|#luxury|"
        ]
    }
}
```

These dictionary statements define the following specializations of the entity `#car`:

```
#sports#car #race#car #economy#car #luxury#car
```

The same can be done for attributes:

```
{
    "|<#email>|": {
        "/$is()": [
            "|<#home>|",
            "|<#work>|",
            "|<#school>|",
            "|<#priority>|"
        ]
    }
}
```

These dictionary statements define the following specializations of the attribute `<#email>`:

```
#home#email #work#email #school#email #priority#email
```

Not all XDI dictionary definitions are generic, i.e., at the top level. The reason for the “X” in XDI (“extensibility”) is that any XDI authority may define its own XDI vocabulary in its own XDI namespace. To do this, the enclosing pipe `| |` syntax is used with an XDI authority (personal or grooup) identifier to define a *dictionary space*. Following is an example of the `<#email>` attribute being specialized by the group `+(https://xdi.org/)` in its own dictionary space:

```
{
    "|<+(https://xdi.org/)>||<#email>|": {
        "/$is#": [
            "|<$string>|"
        ]
    },
    "|<+(https://xdi.org/)>||<#email>|<$xbnf>": {
        "&": "1*( ALPHA / DIGIT ) %22@xdi.org%22"
    }
}
```

Note that `$xbnf` is an XDI-addressable variant of BNF that will be defined in the XDI Dictionary 1.0 specification. This specialized dictionary definition may now be used in XDI statements by prefixing the `<#email>` attribute with the `|+(https://xdi.org/)|` attribute dictionary space. For example:

```
{
    "=example": {
        "|<+(https://xdi.org/)>|<#email>": {
            "&": "example@xdi.org"
        }
    }
}
```

In English the same type of specialization would be expressed by prefixing an entity or attribute name with a proper noun. For example, the generic concept of an “email address” could be specialized by calling it an “XDI.org email address”.

9 Variables { }

A variable is an XDI context node that represents a set of XDI context nodes that will replace the variable context node when it is instantiated. Variables are needed in XDI policies, queries, and other expressions that need to reference a set of XDI nodes that meet specified parameters. The XDI identifier of a variable context node MUST be enclosed with curly brackets `{ }`.

When instantiated, a variable is matched against a target graph. The matching rules depend on the type of variable. All variables except the common variable MUST match at least one arc to have a match. The common variable always has a match because it can also match zero arcs.

9.1 The Common Variable

The common variable is the simplest of all XDI variables. Like the common root node, it is empty, consisting of only a pair of curly brackets. By definition, the common variable matches any subgraph rooted in the context node where the common variable appears. The common variable MAY match any number of arcs, including zero. For example, the following XDI operation will return the entire subgraph rooted in `=example<#home>`:

```
.../$get/=example<#home>{}
```

9.2 Typed Variables

A typed variable is a variable containing an XDI class identifier. The matching rules for typed variables are:

1. A matching instance of a typed variable MUST be a member of the XDI class identified by the variable.

2. By default, a matching instance MAY be at any depth in the subgraph rooted on the typed variable. (To constrain the depth of matching requires using a defined variable—see Defined Variables).

3. A typed variable containing a singleton class identifier MUST match either a singleton instance of that class or a member instance of a collection of that class.

4. A typed variable containing a collection class identifier MUST match an instance of that collection class.

Examples:

Table 12. Typed Variables

Variable	Matches an instance of
{()}	A peer root node
{(/)}	An inner root node
{=}	A person
{+}	A group
{*}	A thing
{@}	An ordinal
{[]}	An entity collection
{<>}	An attribute
{[<>]}	An attribute collection
{<*>}	An attribute instance
{<@>}	An attribute ordinal
{$}	A reserved class
{#}	An unreserved class
{#vehicle}	A specific unreserved class—#vehicle
{[$]}	A reserved entity collection
{[$to]}	A specific reserved entity collection—$to
{[#]}	An unreserved entity collection
{[#vehicle]}	A specific unreserved entity collection—$vehicle
{<$>}	A reserved attribute
{<$iri>}	A specific reserved attribute—$iri
{<#>}	An unreserved attribute
{<#email>}	A specific unreserved attribute—#email
{[<$>]}	A reserved attribute collection
{[<$iri>]}	A specific reserved attribute collection—$iri
{[<#>]}	An unreserved attribute collection
{[<#email>]}	A specific unreserved attribute collection—$email

9.3 Defined Variables

A defined variable is a variable that has additional constraints defined by a set of XDI statements using XDI dictionary vocabulary. The syntax for defined variables is the same as for typed variables except the class identifier uses XDI definition syntax, i.e., is enclosed in pipe characters. Examples:

```
{|=|}
{|+|}
{|*|}
{|#vehicle|}
{|<$iri>|}
{|<#email>|}
```

Note that the common variable may also serve as a defined variable by including a pair of pipe characters inside the curly brackets, i.e.:

```
{||}
```

In this case the defined variable itself imposes no type constraint on the instance; all constraints will be defined by the variable definition.

The location of the definition of a defined variable depends on the location of the defined variable.

1. If the defined variable appears in an XDI subject, the definition MUST appear in the same context as the defined variable.

2. If the defined variable appears in an XDI object, the definition MUST appear in the same context as the defined variable within an inner graph of that XDI statement.

The defined variable definition MUST be expressed using additional XDI statements as defined in the XDI Dictionary specification. For example, in the following XDI statement, the typed variable `{<#phone>}` will match any instances of a phone number to any depth below `=example`.

```
.../$get/=example{<#phone>}
```

To constrain: a) the match to only one instance, and b) the maximum depth of matching nodes to three, use the defined variable `{|<#phone>|}` with two definition statements in an inner graph:

1. The first definition statement constrains the number of matches using the XDI dictionary cardinality attribute <$n>.

2. The second definition statement constrains the depth of matches using the XDI dictionary attribute <`$depth`>.

Example:

```
{
    "...": {
        "/$get": [
            "=example{|<#phone>|}"
        ]
    },
    "(.../$get)": {
        "=example": {
            "|<#phone>|<$n>": {
                "&": "1"
            },
            "|<#phone>|<$depth>": {
                "&": "1-3"
            }
        }
    }
}
```

Note: when there are more matches for a variable than a definition constraint allows, it is up to the XDI endpoint (or the XDI authority for queried graph) to select the matches.

9.4 Reserved Variables

Reserved variables are typed variables that use an XDI reserved class name (keyword) and have a specified function in the XDI protocol. Examples:

```
{$to}
{$from}
{$do}
```

Reserved variables are not defined variables because their definitions are not expressed as a set of XDI statements. Instead their function is specified in one or more XDI specifications, e.g., XDI Messaging, XDI Link Contracts, XDI Discovery, etc. [**XDI-Messaging-V1.0**]

9.5 Metavariables

Variables are commonly used in XDI templates— XDI subgraphs used as the model for creating other XDI subgraphs. When an XDI template is instantiated, the variables it contains are replaced with a valid instance of each variable.

In some cases, an XDI template must contain a variable whose instantiation will be another variable. To enable this, XDI supports metavariables—one variable that contains another. Metavariables are expressed with a double pair of curly brackets. The outer pair represents the containing variable; the inner pair represents the metavariable. Following is an example of a reserved metavariable:

```
{{$from}}
```

To avoid recursion, variables may only be nested one level deep.

9.6 The Literal Variable

XDI templates may need a variable to be instantiated with a literal value. This is called a literal variable and is expressed using the XDI literal symbol: `{&}`.

The literal variable MUST only be used as an XDI predicate. The XDI object of this predicate MUST be an attribute variable. When instantiated, the attribute variable MUST be instantiated with a valid attribute value.

Following is an example of an XDI template that uses a literal variable to specify that a minimum age value is required when the template is instantiated.

```
{
    "{{$to}}": {
        "<#minimum><#age>": {
            "/{&}": [
                "{<#age>}"
            ]
        }
    }
}
```

Following is an example instantiation of this template.

```
{
    "{$to}": {
        "<#minimum><#age>": {
            "&": 13
        }
    }
}
```

10 Core Relations

This section defines the set of XDI relations that express standard relationship types in a semantic graph. These together with the XDI ABNF rules define the universal grammar of XDI. As shown in thetable below, many of the core relations correspond to the basic types of relationships defined in UML [**uml**] and other object modeling languages.

Table 13. Summary of Core Relations

Category	Relationship	Expresses	Relation name	Predicate	Inverse
Equivalence	Transitive equivalence	Logical union of subject and object nodes	Identity relation	/$is/	/$is/
	Visible canonical equivalence	Transpose subject node onto object node	Reference relation	/$ref/	/isref/
	Hidden canonical equivalence	Transpose object node onto subject node	Replacement relation	/$rep/	/isrep/
Hypernym-hyponym; supertype-subtype; superclass-subclass	Subsumption; is-a, type-of	Taxonomic hierarchy; inheritance	Type relation (between classes)	/#/	/$is#/
Class-instance; concept-object; type-token	Instantiation; instance-of	Instance belongs to a class	Type relation (between class and instance)		
Holonym-meronym	Aggregation; has-a	Possession without ownership	Aggregation relation	/$has/	/ishas/
	Composition; part-of	Possession with ownership	Contextual relation	//	/$is()/
	Containment; member-of	Set membership	Collection relation		

10.1 Equivalence Relations

The same resource may be represented by multiple context nodes within an XDI graph. The XDI graph model provides two ways such equivalence may be asserted:

1. Equivalent identifiers may be used in different XDI contexts.

2. Equivalence statements may be made between different XDI contexts.

Equivalence statements are made using using three types of equivalence relations:

1. An identity relation asserts that two context nodes represent the same logical resource and that neither context is canonical. In this case a complete description of the resource requires a union of both subgraphs.

2. A reference relation asserts that two context nodes represent the same logical resource and the object node is canonical. In this case only the object node may contain a subgraph describing the resource.

3. A replacement relation is the same as a reference relation except that XDI address of the object node is replaced by the XDI address of the subject node. In this case it will appear as if the subject node is canonical, i.e., the object subgraph will appear as if it was the subject subgraph, and the replacement relation will be invisible to a requestor.

10.1.1 Equivalent Identifiers

The first way equivalence may be established between two context nodes is by using the same absolute XDI identifier to identify the final arc in the context path. If two XDI context node addresses terminate in the same absolute XDI identifier, those XDI addresses MUST represent the same logical resource in different contexts. The two nodes MAY also have an explicit equivalence relation, but such a relation is not required to establish equivalence.

Absolute XDI identifiers are defined in *Absolute and Relative Identifiers* section.

Following is an example of the same natural person represented by the same absolute XDI name `=alice` in three different contexts (the common root context and two group contexts).

```
{
    "=alice": {
        "<#email>": {
            "&": "alice@example.name"
        }
    },
    "+example-company=alice": {
        "<#email>": {
            "&": "asmith@example-company.com"
        }
    },
    "+example-club=alice": {
        "<#email>": {
            "&": "alice.smith@example-club.com"
        }
    }
}
```

Since XDI names are mutable (reassignable), immutable references require XDI numbers. Here is the same example using XDI numbers in the form of UUIDs.

```
{
    "=!:uuid:33ad7beb-1abc-4a26-b892-466df4379a51": {
        "<#email>": {
            "&": "alice@example.name"
        }
    },
    "+!:uuid:f336a645-f5a9-41b7-ab80-ace41a8f69c2=!:uuid:33ad7beb-1abc-4a26-b892-466df4379
        "<#email>": {
            "&": "asmith@example-company.com"
        }
    },
    "+!uuid:1c958708-d5aa-4213-a6a9-73dd423502b3=!:uuid:33ad7beb-1abc-4a26-b892-466df4379a
        "<#email>": {
            "&": "alice.smith@example-club.com"
        }
    }
```

```
}
```

10.1.2 Identity Relations ($is)

It may not be possible or desirable to use the same absolute XDI identifier for the same resource in different XDI contexts. In this case the same type of equivalence MAY be established using an identity relation. An identity relation MUST be expressed using the XDI predicate `$is`.

Unlike all other XDI predicates, there is no inverse form—`$is` is its own inverse. See *Inverse Relations*.

A `$is` assertion of equivalence is reflexive, symmetric and transitive. It is not canonical, meaning that both the subject node and object node MAY have their own subgraphs without restriction. This means a full description of the described resource requires a union of both subgraphs.

A `$is` statement does not have any special XDI processing rules. Therefore if an XDI endpoint returns a `$is` statement, it is the requestor's responsibility to determine if it needs to request the subgraph identified by the object of that statement.

Following are two examples of `$is` identity relations. The first asserts equivalence between two XDI names —a rooted absolute person name and a nested relative personal name (in the context of a rooted absolute group name).

```
{
    "=alice": {
        "/$is": [
            "+example.club=~alice.smith"
        ]
    }
}
```

The second asserts equivalence between two XDI numbers (in the form of UUIDs)—a rooted absolute person number and a nested absolute person number (in the context of a rooted absolute group number).

```
{
    "=!:uuid:33ad7beb-1abc-4a26-b892-466df4379a51": {
        "/$is": [
            "+!:uuid:1c958708-d5aa-4213-a6a9-73dd423502b3=!:uuid:8ded2f7e-afb4-4494-9918-0
        ]
    }
}
```

10.1.2.1 Replacement Relations ($rep)

A replacement relation is identical to a reference relation except for the XDI addressing rules. With a reference relation, the XDI address of the object node is canonical—the `$ref` relation on the subject node "redirects" to the object node. With a replacement relation, the XDI address of the subject node is canonical, and the subgraph of the object node is logically transposed onto the subject node. The XDI address of the object node is never revealed—nor is the replacement relation. It is only visible to the XDI authority for the graph. A replacement relation MUST be expressed using the XDI predicate `$rep`. An inverse replacement relation MUST be expressed with the XDI predicate `$is$rep`. See Inverse Relations.

A `$rep` assertion of equivalence is irreflexive, asymmetric and transitive. Because it is canonical, a context node described by a `$rep` relation MUST be the subject of exactly one `$rep` statement and MUST NOT be the subject of any other XDI statements.

An inverse replacement relation is not canonical, so a context node described by an `$is$rep` relation MAY be the subject of more than one `$is$rep` statement and MAY contain its own subgraph.

When a $rep relation is reached while traversing an XDI address, the $rep relation MUST be followed to the object node, and traversal of the XDI address MUST continue from the object node. When an XDI operation requests a subgraph containing a $rep relation, the $rep relation MUST NOT be included in the returned subgraph. Instead the object's subgraph MUST be transposed to become the subject's subgraph.

Replacement relations are fundamental to Privacy by Design [**pbd**] [http://privacybydesign.ca/]. In particular, $rep relations enable XDI authorities to publish pseudonyms in order to control correlation between different XDI contexts. Following is an example of a private XDI number used as a pseudonym to share a person's age using a $rep relation without revealing the individual's public XDI number. See *Public and Private Identifiers*.

```
{
    "=!:uuid:33ad7beb-1abc-4a26-b892-466df4379a51": {
        "<#age>": {
            "/$rep": [
                "+!:uuid:1c958708-d5aa-4213-a6a9-73dd423502b3<#age>"
            ]
        }
    },
    "+!:uuid:1c958708-d5aa-4213-a6a9-73dd423502b3": {
        "<#name>": {
            "&": "Alice Smith"
        },
        "<#email>": {
            "&": "alice@example.com"
        },
        "<#age>": {
            "&": 33
        }
    }
}
```

If the following XDI $get request was performed on the graph above:

```
.../$get/=!:uuid:33ad7beb-1abc-4a26-b892-466df4379a51<#age>
```

Then the following XDI graph would be returned:

```
{
    "=!:uuid:33ad7beb-1abc-4a26-b892-466df4379a51": {
        "<#age>": {
            "&": 33
        }
    }
}
```

10.1.3 Equivalence and Directed Graph Cycles

The contextual arcs in an XDI graph form a rooted tree. To ensure deterministic traversal, this rooted tree must be *acyclic*—it cannot allow a chain of contextual arcs to loop around in a circle, i.e., that start at one node and end up back at that same node. In graph theory, such a loop is called a *cycle*.

Cycles would not be a problem if an XDI rooted tree consisted of only contextual arcs where all the labels were relative (like DNS). However if either: a) equivalent identifiers are used for nodes within the rooted tree, or b) equivalence relations are added between nodes in the rooted tree, then it becomes possible to form a cycle. Therefore, if an XDI agent or an XDI endpoint adds an equivalent identifier or an equivalence statement to an XDI graph, it MUST NOT create a cycle.

10.1.4 Equivalence of Context Types and Roles

For two context nodes to be equivalent, they MUST have the same context type and role. For example, if two context nodes represent different XDI names for the same person, both must be person entity nodes. If two context nodes represent the same email address in different contexts, both must be attributes. If two context nodes represent synonyms in a dictionary, both must be definitions.

10.1.5 Reference Relations ($ref)

In contrast to identity relations, which establish the equivalence of two context nodes where neither is canonical, a reference relation establishes the equivalence of two context nodes where the XDI address for the object node is canonical. In this case only the object node may contain a subgraph—the subject node may only contain a reference to the object node.

A reference relation MUST be expressed using the XDI predicate `$ref`. An inverse reference relation MUST be expressed with the XDI predicate `$is$ref`. See Inverse Relations.

A `$ref` assertion of equivalence is irreflexive, asymmetric and transitive. Because it is canonical, a context node described by a $ref relation MUST be the subject of exactly one $ref statement and MUST NOT be the subject of any other XDI statements.

An inverse reference relation is not canonical, so a context node described by an `$is$ref` relation MAY be the subject of more than one `$is$ref` statement and MAY contain its own subgraph. When a `$ref` relation is reached while traversing an XDI address, the `$ref` relation MUST be followed to the object node, and traversal of the XDI address MUST continue from the object node. By default, when an XDI operation requests a subgraph containing a `$ref` relation, both the `$ref` relation and the object subgraph will be included in the returned subgraph.

Note: this behavior can be modified by an XDI messaging parameter as defined in the XDI Messaging specification.

Reference relations are needed for one of the most common patterns in XDI graphs: mapping a human-friendly mutable (reassignable) identifier (an XDI name) to a machine-friendly immutable (persistent) identifier (an XDI number). This is called the *name/number reference pattern*. The following example shows a name/number `$ref` relation for both a person and a group:

```
{
    "=alice": {
        "/$ref": [
            "=!:uuid:33ad7beb-1abc-4a26-b892-466df4379a51"
        ]
    },
    "+example.club": {
        "/$ref": [
            "+!:uuid:1c958708-d5aa-4213-a6a9-73dd423502b3"
        ]
    }
}
```

10.1.6 Rules for Processing Equivalence Relations

Rules for processing of operations on XDI graphs are defined in the XDI Messaging specification [**XDI-Messaging-V1.0**]. This includes rules for processing of equivalence relations. For informative purposes, here is a summary:

1. A standard XDI `/$get/` operation must return all reference relations but must not return any replacement relations.

2. If an XDI `/$get/` message specifies a `<$deref>` parameter, the operation must process all reference relations as if they were replacement relations. This means: a) each `$ref` predicate in the result graph must be replaced with a `$rep` predicate, and b) the resulting statement must be processed as a replacement relation. Therefore the XDI graph returned by this `/$get/` operation will not contain any reference or replacement relations—they will all have been dereferenced.

3. If a reference relation or replacement relation is included in the processing of an XDI `$add`, `$set` or `$del` operation, it must be processed according to the standard rules for equivalence relations, and processing of the operation must continue at the object of the statement. The final target statement(s) to be added or deleted must be determined only after all reference or replacement relations have been processed.

4. If the target of an XDI operation is itself a reference relation or replacement relation, that relation must not be processed. Instead, the operation must apply directly to the `$ref` or `$rep` statement.

10.2 Inverse Relations

Due to its semantic tree architecture, the inverse of any XDI predicate may be expressed algorithmically by prefixing it with the `$is` identity equivalence relation. If an XDI predicate is prefixed by \$is, it MUST be interpreted as expressing the inverse relation of the same XDI predicate without the prefix. Examples:

Table 14. Inverting relations with \$is

Subject	Predicate	Object
`=luke.skywalker`	`#father`	`=darth.vader`
`=darth.vader`	`$is#father`	`=luke.skywalker`
`[#vehicle]*!:uuid:x-1`	`#owner`	`=example`
`=example`	`$is#owner`	`[#vehicle]*!:uuid:x-1`
`=example.club`	`$ref`	`+!:uuid:x-2`
`+!:uuid:x-2`	`$is$ref`	`=example.club`

In semantic terms, the `$is` prefix expresses that the subject of a statement whose predicate includes the `$is` prefix is equivalent to the object of the same statement made without the `$is` prefix.

A predicate that uses the `$is` prefix is called an *inverse predicate*. A statement that uses an inverse relation is called an *inverse statement*. A predicate that does not use the `$is` prefix is called a *direct predicate*. A statement that does not use an inverse relation is called a *direct statement*. An inverse relation may be used to express the inverse of any XDI statement except the `$is` identity equivalence relation itself (which is its own inverse). This includes XDI contextual statements. However, because the predicate of an XDI contextual statement is empty, an inverse contextual statement MUST use the predicate `$is()`. The empty parentheses encapsulate the empty predicate.

Inverse contextual statements are particularly useful in XDI graphs because they provide a mechanism for discovering other contexts for a resource. For example, a requestor who only has knowledge of `=alice` could query the following XDI graph (assuming the requestor has permission) to discover that `=alice` also exists in the `+example.company` and `+example.club` contexts.

```
{
    "=alice": {
        "/$is()": [
            "+example-company",
            "+example-club"
        ],
        "<#email>": {
            "&": "alice@example.name"
        }
    },
    "+example-company=alice": {
        "<#email>": {
            "&": "asmith@example-company.com"
        }
    },
    "+example-club=alice": {
        "<#email>": {
            "&": "alice.smith@example-club.com"
        }
    }
}
```

10.3 Type Relations (#)

The heart of ontologies is type relationships, i.e., defining subtypes (subclasses) and supertypes (superclasses). In XDI, these relationships are expressed using XDI type statements. Since the concept of "type" is already represented in XDI as # (the context symbol for unreserved classes), an XDI type relation MUST be expressed using the XDI predicate `#`. An inverse type relation MUST be expressed with the XDI predicate `$is#`.

XDI type statements fall into two categories:

1. **Relationships between subclasses and superclasses.** These correspond to `rdf:subClassOf` relations.

2. **Relationships between instances and classes.** These correspond to `rdf:type` relations.

Since the context symbol for an XDI context already indicates whether it is a class or an instance, the same XDI type relations can express either category of type statement.

Examples of superclass-subclass relations:

Table 15. Superclass-subclass relations

Subject	Predicate	Object
`#food`	`#`	`#fruit`
`#fruit`	`#`	`#apple`
`#fruit`	`#`	`#banana`
`#fruit`	`#`	`#pear`
`<$string>`	`#`	`<#name>`
`<$string>`	`#`	`<#email>`

Examples of subclass-superclass relations:

Table 16. Subclass-superclass relations

Object	Predicate	Object
#fruit	$is#	#food
#apple	$is#	#fruit
#banana	$is#	#fruit
#pear	$is#	#fruit
<#name>	$is#	<$string>
<#email>	$is#	<$string>

Examples of class-instance relations:

Table 17. Class-instance relations

Subject	Predicate	Object
#carpenter	#	=!:uuid:...
#church	#	+!:uuid:...
#car	#	*!:uuid:...

Examples of instance-class relations:

Table 18. Instance-class relations

Object	Predicate	Object
=!:uuid:...	$is#	#carpenter
+!:uuid:...	$is#	#church
*!:uuid:...	$is#	#car

10.4 Aggregation Relations ($has)

From an object modeling standpoint, XDI contextual relations are analogous to UML *composition* (whole/part) relationships, where a composite object both possesses and owns a set of component objects. The defining feature of composition is ownership, i.e., when a composite object is destroyed, so are its component objects. This is true of XDI contexts—if a context node is deleted, so are all the context nodes in its subgraph.

UML also defines *aggregation* relationships, where an aggregate object possesses but does not own a set of aggregated objects. The defining feature of aggregation is lack of ownership, i.e., when an aggregate object is destroyed, its aggregated objects are not. They continue to live independently of the aggregate object.

This is often called a “has-a” relationship. For this reason, aggregation relations in XDI are expressed using the XDI predicate `$has`. An inverse aggregation relation MUST be expressed with the XDI predicate `$is $has`.

Compared to contextual relations, the defining feature of `$has` relations in XDI is the same as in UML: if the subject of a `$has` relation is deleted, the object of the relation is not affected (unless the object happens to be in the subgraph of the subject).

A classic example is a university department: it has a set of professors, but if the university department is closed, the professors still exist. The following example shows both the university group entity and each of the professors’ person entities as independent root-level entities aggregated via a `$has` relation:

```
{
    "+example-university#magic#department": {
        "/$has": [
            "=example-prof-1",
            "=example-prof-2",
            "=example-prof-3"
        ]
    },
    "=example-prof-1": {
        "<#name>": {
            "&": "Albus Dumbledore"
        }
    },
    "=example-prof-2": {
        "<#name>": {
            "&": "Minerva McGonagall"
        }
    },
    "=example-prof-3": {
        "<#name>": {
            "&": "Severus Snape"
        }
    }
}
```

Because the aggregated entities in this example are root-level, their attributes (such as the `<#name>` attribute shown above) describe them independently of any other context. XDI's semantic tree architecture also enables the aggregated entities to be described in the context of the aggregating entity. This is called the *contextual description pattern*. The advantage of contextual descriptions is that the attributes and relations of the aggregated entity can be specific to the aggregating context.

For example, in the university professor scenario above, contextual descriptions may be used to express the names and email addresses of the university professors in the context of a specific university department.

```
{
    "+example-university#magic#department=example-prof-1": {
        "<#name>": {
            "&": "Headmaster Dumbledore"
        },
        "<#email>": {
            "&": "adumbledore@magic.example.edu"
        }
    },
    "+example-university#magic#department=example-prof-2": {
        "<#name>": {
            "&": "Professor McGonagall"
        },
        "<#email>": {
            "&": "mmcgonagall@magic.example.edu"
        }
    },
    "+example-university#magic#department=example-prof-3": {
        "<#name>": {
            "&": "Professor Snape"
        },
        "<#email>": {
            "&": "ssnape@magic.example.edu"
        }
    }
}
```

Contextual description of the members of a group is optional because it is only needed when a group member has context-specific attributes or relations. When it is required to express group membership in XDI, it MUST be expressed using a `$has` relation between the group entity and each group member entity. A group member entity MAY be either a person entity or another group entity, i.e., groups may contain other groups.

In general, as with UML aggregation relationships, an XDI `$has` aggregation relation does not constrain the type of entity that may be aggregated. There is one exception. If the subject of a `$has` relation is a collection, the object of this relation MUST be a member of the collection class. This is called the *virtual collection pattern*. It applies whenever members of a collection are independent entities that do not have contextual descriptions. A typical example is a person’s music album collection.

```
{
    "=example-person[#album]": {
        "/$has": [
            "+moody-blues[#album]*~every-good-boy-deserves-favour",
            "+rolling-stones[#album]*~sticky-fingers",
            "+derek-and-the-dominos[#album]*~layla"
        ]
    },
    "+moody-blues[#album]*~every-good-boy-deserves-favour": {
        "<#release><#year>": {
            "&": "1971"
        }
    },
    "+rolling-stones[#album]*~sticky-fingers": {
        "<#release><#year>": {
            "&": "1971"
        }
    },
    "+derek-and-the-dominos[#album]*~layla": {
        "<#release><#year>": {
            "&": "1971"
        }
    }
}
```

Virtual collections are another example of how traditional database indexes may be modeled in an XDI graph.

`$has` aggregation relations are as useful in XDI as they are in UML. They can model many forms of resource ownership and control. A specific application is digital signatures: an XDI signature attribute can use a `$has` relation to specify the set of XDI subgraphs included in the signature’s scope. The use of `$has` relations with XDI digital signatures is specified in the XDI Cryptography specification [**XDI-Cryptography-V1.0**].

10.5 Boolean Relations

To meet the design goal of describing authorization, policy, and rights expression, XDI must be able to describe Boolean logic trees. These relations are formally defined in the XDI Link Contracts specification but are summarized below for reference:

Table 19. Boolean Relations

Relation	Definition
$true	Logical true statement
$false	Logical false statement
$and	Logical conjunction
$or	Logical disjunction
$not	Logical negation
$if	Logical branching

10.6 XDI Operations

The final category of core relations is the set of XDI predicates representing the standard XDI protocol operations on XDI graphs. These are formally defined in the XDI Messaging specification [**XDI-Messaging-V1.0**], but are summarized below for reference:

Table 20. Boolean Relations

Relation	Operation in Target Graph
$get	Read statement
$set	Write statements
$add	Add new statements
$mod	Update value
$del	Delete statements
$connect	Instantiate a new XDI connection
$send	Send an XDI message
$push	Publish an XDI message to subscribers
$do	Authorize other operations

10.7 Nominalization

In linguistics, nominalization is the use of a verb as a noun. In XDI, nominalization is the use of an XDI relation as an XDI context. The *nominalization pattern* is needed in XDI for the same reason as in human language: to embody relations so they may have their own descriptions, attributes, and ordering.

A common case is when an XDI protocol operation needs a parameter. For example, in the following XDI message, the XDI `$get` request in the first statement is nominalized in order to add the `<$deref>` parameter in the second statement.

```
{
   "=!:uuid:x-1[$msg]*!:uuid:x-2$do": {
       "/$get": [
           "=!:uuid:x-3<#email>"
       ]
   },
   "=!:uuid:x-1[$msg]*!:uuid:x-2$do$get": {
       "<$deref>": {
           "&": true
       }
   }
}
```

Another common example involves the name/number reference pattern, where a mutable human-friendly XDI name for an entity has a `$ref` equivalence relation to an immutable XDI number for the entity. With this pattern it is easy to start with the XDI name to look up the XDI number. However a relying party such as an online merchant may frequently need to do the opposite, i.e., look up a human-friendly XDI name from a persisted XDI number (e.g., to greet a returning customer).

This is not as simple as following an inverse reference relation (`$is$ref`) from the XDI number to the XDI name because the person may have more than one XDI name that references his/her XDI number.

As shown below, the solution is to nominalize the `$is$ref` relation so it may be treated as a subcontext in the person’s XDI graph. This new subcontext can now express a canonical `ref` relation to the person’s preferred XDI name. (Note that this preference is stored inside an inner graph describing the merchant’s unique relationship with that particular customer.)

```
{
    "=!:uuid:x-1": {
        "/$is$ref": [
            "=alice",
            "=alison",
            "=alice.smith"
        ]
    },
    "+!:uuid:x-2": {
        "/$is$ref": [
            "+example.merchant"
        ]
    },
    "(+!:uuid:x-2/=!:uuid:x-1)": {
        "=!:uuid:x-1$is$ref": {
            "/$ref": [
                "=alice.smith"
            ]
        }
    }
}
```

11 ABNF

An XDI graph in *statement format*, where there is one XDI statement for each arc in the graph, MUST be valid according to the ABNF rules in this section. The *Serialization* section specifies the rules for serializing a valid XDI graph in JSON.

Note that the ABNF rules alone do not express all requirements for an XDI graph to be semantically valid. Additional rules for semantic validity are stated in text both within this section and in other sections.

11.1 XDI Graph

```
xdi-graph                = *( xdi-statement / CRLF )
xdi-statement            = contextual-statement / literal-statement / relational-statement
```

In statement format, an XDI graph is a set of XDI statements consisting of a sequence of Unicode characters. It is intended only for encoding in Unicode encodings such as UTF-8 and UTF-16. Encoding in other encodings may result in character corruption and unpredictable results.

All statements are one of three types: contexual, literal, or relational.

11.1.1 Contextual Statements

```
contextual-statement     = direct-contextual / inverse-contextual

direct-contextual        = peer-root-direct / inner-root-direct / entity-direct / attr-dire
peer-root-direct         = *peer-root "//" peer-root
inner-root-direct        = root-address "//" inner-root
entity-direct            = entity-address "//" entity
attr-direct              = attr-address "//" attr

inverse-contextual       = peer-root-inverse / inner-root-inverse / entity-inverse / attr-i
peer-root-inverse        = peer-root   "/$is()/" *peer-root
inner-root-inverse       = inner-root   "/$is()/" root-address
entity-inverse           = entity "/$is()/" entity-address
attr-inverse             = attr   "/$is()/" attr-address
```

Contextual statements define the branch nodes of a semantic tree. See *Contextual Arcs and Contextual Statements*. A direct contextual statement MAY have an inverse contextual statement. An inverse contextual statement MUST have an direct contextual statement. Note that a direct contextual statement MUST have exactly one XDI context node as an object, and an inverse contextual statement MUST have exactly one XDI context node as a subject.

11.1.2 Literal Statements

```
literal-statement        = entity-address 1*attr "/&/" value
literal-var-statement    = entity-address 1*attr "/{&}/" value-variable
value-variable           = "{" attr-class "}"
```

Literal statements define the leaf nodes of a semantic tree. See *Literal Arcs and Literal Statements*. An additional rule applies to literal statements: the final node sequence before a literal predicate MUST be one of two options:

1. An attribute class (`attr-class`).
2. An attribute collection (`attr-collection`) followed by an attribute instance (`attr-instance`).

11.1.3 Relational Statements

```
relational-statement    = direct-relational / inverse-relational / relation-definition
direct-relational       = xdi-address "/" 1*entity "/" xdi-address
inverse-relational      = xdi-address "/$is" 1*entity "/" xdi-address
```

Relational statements define relationships between context nodes in a semantic tree. See *Relational Arcs and Relational Statements*. A direct relational statement MAY have an inverse relational statement. An inverse relational statement MAY have a direct relational statement. The existence of a direct relational statement does not require the existence of its inverse, and vice versa.

11.1.4 Relation Definition Statements

```
relation-definition     = direct-domain / inverse-domain / direct-range / inverse-range

direct-domain           = direct-entity-domain / direct-attr-domain
direct-entity-domain    = root-address 1*definition "/(/)/" root-address 1*definition
direct-attr-domain      = root-address *definition 1*attr-definition "/(/)/" root-address

inverse-domain          = inverse-entity-domain / inverse-attr-domain
inverse-entity-domain   = root-address 1*definition "/$is(/)/" root-address 1*definition
inverse-attr-domain     = root-address 1*definition "/$is(/)/" root-address *definition 1*

direct-range            = direct-entity-range / direct-attr-range
direct-entity-range     = root-address 1*definition "/(/)#/" root-address 1*definition
direct-attr-range       = root-address 1*definition "/(/)#/" root-address *definition 1*at

inverse-range           = inverse-entity-range / inverse-attr-range
inverse-entity-range    = root-address 1*definition "/$is(/)#/" root-address 1*definition
inverse-attr-range      = root-address *definition 1*attr-definition "/$is(/)#/" root-addr
```

In XDI dictionaries, relational statements are used to define the domain and range of XDI predicates. The empty inner graph `(/)` is the relation definition predicate.

11.2 XDI Address

The arcs of an XDI address must follow this order: first any peer roots, then any inner roots, then any entity arcs, then any attribute arcs, then the literal symbol `&` (if the address identifies a literal node). If the address terminates in the literal symbol, it MUST be preceded by either an attribute class or the combination of an attribute collection and attribute instance as stated in the Literal Statements section above. The required sequence is summarized in the following ABNF rule, which however is not suitable for deterministic parsing because of the preceding restriction, so we do not list it as normative.

```
xdi-address = *peer-root *inner-root *entity *attr [ attr "&" ]
```

XDI addresses are categorized for use in Contextual Statements:

```
xdi-address             = root-address / entity-address / attr-address / literal-address
```

```
root-address            = *peer-root *inner-root
entity-address          = root-address *entity
attr-address            = entity-address *attr
literal-address         = entity-address 1*attr "&"
```

11.3 XDI Contexts

The rules in this section define the valid syntax for the three different primary roles for XDI context nodes.

11.3.1 Root Contexts

```
peer-root               = peer-root-instance / peer-root-variable
peer-root-instance      = "(" entity ")"
peer-root-variable      = "{" peer-root-instance "}"

inner-root              = inner-root-instance / inner-root-variable
inner-root-instance     = inner-root-peer-root / inner-root-entity
inner-root-peer-root    = "(" *peer-root "/" *entity ")"
inner-root-entity       = "(" *entity "/" *entity ")"
inner-root-variable     = "{" inner-root-instance "}"
```

Both peer root context nodes and inner root context nodes are syntactically distinguished by enclosing parentheses. Inner root context nodes contain the subject and predicate of the XDI relational statement that defines the inner root graph.

11.3.2 Entity Contexts

```
entity                  = singleton / collection / definition / variable / meta-variable
singleton               = instance / class
collection              = "[" class "]"
definition              = "|" ( singleton / collection ) "|"
variable                = "{" ( singleton / collection / definition ) "}"
meta-variable           = "{" variable "}"

instance                =  person / group / thing / ordinal
person                  = "=" [ "!" ] [ "~" ] id-string
group                   = "+" [ "!" ] [ "~" ] id-string
thing                   = "*" [ "!" ] [ "~" ] id-string
ordinal                 = "@" [ "!" ] [ "~" ] ordinal-string

class                   = reserved-class / unreserved-class / "$" / "#" / "=" / "+" / "*"
reserved-class          = "$" xdi-name
unreserved-class        = "#" [ "~" ] id-string
```

Entity context nodes are either classes or instances. Class node identifiers are always immutable, so they do not use the immutability symbol `!`. Instance node identifiers may use both the immutability symbol and the relativity symbol ~.

11.3.3 Attribute Contexts

```
attr                    = attr-singleton / attr-collection / attr-definition / attr-variab
```

```
attr-singleton          = attr-class / attr-instance
attr-collection         = "[" attr-class "]"
attr-definition         = "|" ( attr-singleton / attr-collection ) "|"
attr-variable           = "{" ( attr-singleton / attr-collection / attr-definition ) "}"
attr-meta-variable      = "{" attr-variable "}"
attr-class              = "<" class ">"
attr-instance           = "<" instance ">"
```

The rules for attribute context nodes are very similar to the rules for entity context nodes, with the addition of chevron brackets < and > to indicate the attribute role.

11.4 XDI Identifiers

```
id-string               = xdi-name / xdi-scheme / encap-iri
ordinal-string          = int / other-scheme
```

For all XDI identifiers except ordinals, there are three basic identifier types: names/numbers, schemes, and encapsulated IRIs. Ordinals are restricted to either integers or another scheme with an associated specification for defining the order of identifiers under that scheme. See *XDI Schemes*.

11.4.1 XDI Names and Numbers

```
xdi-name                = ID_Start *( ID_Continue / "_" / "-" / "." )
pct-encoded             = "%" HEXDIG HEXDIG
```

Note that the ABNF rule `xdi-name` covers both XDI names and XDI numbers as defined in *XDI Names and Numbers*.

An XDI identifier MUST start with a character with the ID_START property defined by the Unicode Identifier and Pattern Syntax [**UAX31**]. The characters following the ID_START character MUST have the ID_CONTINUE property or be underscore _, hyphen `-`, or period `.`.

For compatibility, users SHOULD define and enter XDI names and numbers using lower case as a normalization to avoid interoperability problems with other case-insensitive systems. See *Normalization and Comparison*.

11.4.2 XDI Schemes

```
xdi-scheme              = uuid-scheme / cid-scheme / other-scheme
uuid-scheme             = ":uuid:" 8HEXDIG "-" 4HEXDIG "-" 4HEXDIG "-" 2HEXDIG 2HEXDIG "-"
cid-scheme              = ":cid-" 1*DIGIT ":" xdi-name
other-scheme            = ":" ( lower-alpha / DIGIT ) *( lower-alpha / DIGIT / "_" / "-" /
```

See *XDI Schemes*.

11.4.3 Encapsulated IRIs

The rules in this section have been simplified from [**RFC 3987**] as sufficient for XDI processors to recognize IRI syntax without intensive verification.

```
encap-iri               = "(" absolute-iri ")"
```

```
absolute-iri            = iri-scheme ":" 1*iri-char
iri-scheme              = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )
iri-char                = safe-char / %xA0-EFFFD / pct-encoded
safe-char               = unreserved / reserved / gen-delims / safe-sub-delims
unreserved              = ALPHA / DIGIT / "-" / "." / "_" / "~"
reserved                = gen-delims / safe-sub-delims
gen-delims              = ":" / "/" / "?" / "#" / "[" / "]" / "@"
safe-sub-delims         = "!" / "$" / "&" / "(" / "*" / "+" / "," / ";" / "="
```

Note that, in comparison to the `sub-delims` rule in RFC 3987, the `safe-sub-delims` rule has removed:

1. The right parentheses character `)` in order to prevent it from being interpreted as terminating the encapsulated IRI.

2. The single quote character `'` in order to prevent it from being interpreted as terminating a JSON string.

Additional rules for this section:

1. XDI processors are NOT REQUIRED to check URIs for valid URI syntax.

2. XDI parsers reading an URI (i.e. having found an open parenthesis followed by a string matching uri-scheme followed by a colon) MAY simply consider following characters part of the URI up to the close parenthesis `)`.

11.5 Rules Inherited from JSON

The rules in this section covering JSON primitives are adapted from and equivalent to those in [**RFC 7159**].

```
value                   = "false" / "null" / "true" / object / array / number / string
object                  = begin-object [ member *( value-separator member ) ] end-object
member                  = string name-separator value
array                   = begin-array [ value *( value-separator value ) ] end-array

begin-array             = ws %x5B ws  ; [ left square bracket
begin-object            = ws %x7B ws  ; { left curly bracket
end-array               = ws %x5D ws  ; ] right square bracket
end-object              = ws %x7D ws  ; } right curly bracket
name-separator          = ws %x3A ws  ; : colon
value-separator         = ws %x2C ws  ; , comma

number                  = [ "-" ] int [ frac ] [ exp ]
exp                     = [ "e" / "E" ] [ "-" / "+" ] 1*DIGIT
frac                    = "." 1*DIGIT
int                     = "0" / ( %x31-39 *DIGIT )   ; no leading zeros

string                  = quotation-mark *char quotation-mark

char                    = unescaped / backslash ( quotation-mark / backslash /
                          "/" / "b" / "f" / "n" / "r" / "t" / "u" 4HEXDIG )

backslash               = %x5C           ; \ reverse solidus U+005C
quotation-mark          = %x22           ; " quotation mark  U+0022

unescaped               = %x20-21 / %x23-5B / %x5D-10FFFF
```

```
ws                      = *( %x20 / %x09 / %x0A /  %x0D )
```

Additional rules for this section:

1. The ABNF in RFC 7159 MUST be authoritative. The JSON ABNF given here is just a readable equivalent given for the reader's convenience.

2. While RFC 7159 allows whitespace between JSON tokens to be space, tab, CR, or LF, an XDI literal statement MUST be a single line, so CR or LF are not allowed as whitespace in JSON values in XDI literal statements.

3. XDI generators SHOULD produce JSON values without optional whitespace between tokens.

4. For historical reasons, the `\uxxxx` escape sequence expresses an UTF-16 code unit, implying a non-BMP character must be escaped as two code units, and allowing illegal escape sequences such as unpaired surrogates. This is not ideal, but following RFC 7159, these rules are carried forward for backward compatibility. XDI processors MAY reject input with invalid UTF-16 code sequences but are NOT REQUIRED to check.

11.6 Rules Inherited from ABNF (RFC 2234)

These standard ASCII character classes are inherited from [**RFC 2234**].

```
ALPHA                   = %x41-5A / %x61-7A                            ; A-Z, a-z
DIGIT                   = %x30-39                                      ; 0-9
HEXDIG                  = %x30-39 / %x41-46 / %x61-66                  ; 0-9, a-f, A-F
CRLF                    = %x0D / %x0A / ( %x0D %x0A )
```

12 Serialization

In keeping with the design goal of serialization independence, an XDI graph may be serialized in any specified format. JSON was selected as the first format due to its compact structure and minimal overhead. The XDI TC may specify additional serialization formats in the future, including XML (and potentially binary formats).

The JSON serialization format specified in this section was developed to strike a balance between a *completely flat serialization model*—where every XDI address in a graph has its own unique key in a single top-level JSON object—and a *completely nested serialization model*—where every single node in an XDI graph is represented by its own nested JSON object regardless of depth.

The XDI Technical Committee chose a mid-point of four levels of nested JSON objects for the following reasons:

- It directly reflects the four-level XDI root-entity-attribute-literal (REAL) graph model, which in turn reflects the RDF 1.1 quad model.
- It also directly emulates the four components of the RDF 1.1 quad model: graph name, subject, predicate, object.
- This format is easy to navigate and search with tools that support JSON Path [**jsonpath**].
- If you know (or discover) the address of a specific XDI graph, you can easily discover the associated XDI graph relations and XDI entities (assuming you have permission).
- If you know (or discover) the address of specific XDI entity within a graph, you can easily discover the associated XDI entity relations and XDI attributes (assuming you have permission).
- If you know (or discover) the address of specific XDI attribute within a graph, you can easily discover the associated XDI attribute relations and, if present, the literal value of the attribute (assuming you have permission).

12.1 XDI JSON Format Serialization Rules

A valid XDI JSON document MUST be a valid JSON document according to [**RFC 7159**]". Prior to JSON serialization, an XDI graph MUST be valid according to the rules in the ABNF section.

There are two sets of JSON serialization rules depending on whether an XDI agent requests implied contextual statements (`implied=1`) or does not request these statements (`implied=0`, which is the default). See Contextual Arcs and Contextual Statements in *The XDI Graph Model* section. The `implied=1` rules build on the `implied=0` rules.

Definitions:

1. *Solitary context node* is a context node that is only the object of a single contextual statement in the graph, and not the subject or object of any other statements in the graph. Note that an inner root node is *never* a solitary context node because it is always be the object of a relational statement.
2. *Object-only node* is a context node that is the object of one or more relational statements in the graph, but not the subject of any statements in the graph.

12.1.1 Rules When Implied Contextual Statements are Excluded (Implied=0)

12.1.1.1 Root Node Serialization Rules

1. The XDI common graph root MUST be serialized as the top-level JSON object ("common graph object").
2. To serialize a peer or inner graph root node in the common graph, the XDI address of the graph root node MUST be a key string in the common graph object ("graph root key").
3. The JSON value of a graph root key MUST be a second-level nested JSON object ("graph object").
4. If a graph root node is a solitary context node, the graph object MUST be empty.
5. To serialize an XDI graph root relation, the XDI address of the relation MUST be a key string in the graph object ("root relation key") prefixed with a forward slash `/`.
6. The JSON value of a root relation key MUST be an array of the XDI addresses of the XDI object(s) of the relation.

12.1.1.2 Entity Node Serialization Rules

1. To serialize an XDI entity node inside a graph object (i.e., a common graph object, peer graph object, or inner graph object), the XDI address of the entity node MUST be a key string in the graph object ("entity key").
2. The JSON value of an entity key MUST be a nested JSON object ("entity object").
3. If an entity node is a solitary context node, the entity object MUST be empty.
4. To serialize an XDI entity relation, the XDI address of the relation MUST be a key string in the entity object ("entity relation key") prefixed with a forward slash `/`.
5. The JSON value of an entity relation key MUST be an array of the XDI addresses of the XDI object(s) of the relation.

12.1.1.3 Attribute Node Serialization Rules

1. To serialize an XDI attribute node inside a graph object or entity object, the XDI address of the attribute node MUST be a key string in the graph object or entity object ("attribute key").
2. The JSON value of an attribute key MUST be a nested JSON object ("attribute object").
3. If an an attribute node is a solitary context node, the attribute object MUST be empty.
4. To serialize an XDI attribute relation, the XDI address of the relation MUST be a key string in the attribute object ("attribute relation key") prefixed with a forward slash `/`.
5. The JSON value of an attribute relation key MUST be an array of the XDI addresses of the XDI object(s) of the relation.

12.1.1.4 Literal Node Serialization Rules

1. To serialize an XDI literal node containing the value of an XDI attribute, the key string in the attribute object MUST be `&` (the "literal key").

2. The JSON value of the literal key MUST be the JSON value of the XDI literal.

12.1.1.5 Common Variable Node Serialization Rule

1. If an XDI address includes a common variable, that variable MUST be appended to the graph root key, entity key, or attribute key that precedes it.

12.1.2 Rules When Implied Contextual Statements are Included (Implied=1)

These rules MUST be applied in addition to the rules in the previous section.

1. The first-level subcontexts of every graph root node MUST be serialized using the graph root relation `//`.

2. The first-level subcontexts of every entity node MUST be serialized using the entity relation `//`.

3. The first-level subcontexts of every attribute node MUST be serialized using the attribute relation `//`.

4. Every object-only context node MUST be serialized following the same rules as for a solitary context nodes in the previous section.

Note that while the `implied=0` serialization is more verbose, it has two properties that may be useful to thin clients who wish do navigation directly on the returned JSON document.

1. There will be exactly one serialized XDI subject/predicate/object triple (in the common graph) or context/subject/predicate/object quad (in a peer or inner graph) for every arc in the XDI graph. This is the serialization most directly comparable to an [**rdf-datasets**].

    a. Every contextual statement will have the predicate `//`.

    b. Every relational statement will have a predicate that begins with `/`.

    c. Every literal statement will have the JSON object key string &.

2. There will be exactly one JSON object for every XDI context node in the graph, including the outermost JSON object which represents the common root context.

12.2 Readability Rules

For consistent readability, the following rules are RECOMMENDED.

1. At all context levels of an XDI graph, starting with the common root level, XDI statements that are applicable in that context SHOULD appear in the following order:

    a. Implied contextual statements (when `implied=1`).

    b. Relational statements.

    c. Literal statements.

    d. Attribute statements.

    e. Entity statements.

    f. Graph root statements.

2. For a set of XDI statements of the same type at the same level, the JSON objects SHOULD appear in alphabetical order according to the JSON object key string.

3. In a JSON array of XDI addresses, the XDI address values SHOULD appear in alphabetical order.

12.3 JSON Display Format Rules

JSON XDI format is easy enough to read that in most cases there is no need for a separate display format. However there may be cases where either: a) the JSON delimiters may be confusing to a non-technical audience, or b) vertical line space is at a premium. For these situations, we define a simple display version of the JSON XDI format.

1. First, serialize the graph in JSON XDI format as specified above using the conventional JSON display formatting rules where each JSON object and array appears on a new line and is indented one level.

2. Remove all JSON delimiters except those delimiting XDI literal values.

3. Remove all blank lines.

4. Remove one level of indenting.

5. If a line needs to wrap, the wrapped line(s) MUST be indented to the same number of tab stops as the starting line.

6. Any instance of the common root node (which is serialized in the JSON format as the empty string `""`) MUST be replaced by a double forward slash `//`.

Note about this format:

1. The left margin represents the common root node; all subroots, entities, attributes, and relations of the common root node will be aligned with the left margin.

2. All entities, attributes, and relations of a subroot node will be indented one tab stop.

3. All attributes and relations of an entity will be indented one tab stop more than the entity.

4. All relations of an attribute will be indented one tab stop more than the attribute.

5. The ampersand & representing the literal value of an attribute will be indented one tab stop more than the attribute.

6. The JSON value of the attribute will follow the ampersand on the same line, indented one tab stop more than the ampersand.

12.4 Examples

Note: in these examples, UUIDs are shown in a truncated format for readability.

12.4.1 Example Where Implied Contextual Statements are Excluded (Implied=0)

```
{
    "/$is$ref": [
        "(=!:uuid:x-alice)"
    ],
    "<$uri>": {
        "&": "https://xdi.example.com/=!:uuid:x-alice"
    },
    "=!:uuid:x-alice": {
        "/#friend": [
            "=!:uuid:x-bob",
            "=!:uuid:x-carol",
            "(=!:uuid:x-alice/#friend)"
        ],
        "/#spouse": [
            "=!:uuid:x-david"
        ],
        "[<#email>]<@~0>": {
            "&": "alice@example.com"
        },
        "[<#email>]<@~1>": {
            "&": "asmith@example.net"
        },
        "<#email>": {
            "/$ref": [
                "=!:uuid:x-alice[<#email>]<@~0>"
            ]
        },
        "<#home><#email>": {
            "/$ref": [
                "=!:uuid:x-alice[<#email>]<@~1>"
            ]
        },
        "<#work><#email>": {
            "/$ref": [
                "=!:uuid:x-alice[<#email>]<@~1>"
            ]
        }
    },
    "=!:uuid:x-alice#passport": {
        "<#country>": {
            "&": "USA"
        },
        "<#name>": {
            "&": "Alice Smith"
        },
        "<#number>": {
            "&": "1234567"
        }
    },
    "(=!:uuid:x-alice)": {
        "/$ref": [
            ""
        ]
    },
    "(=!:uuid:x-alice/#friend)": {
        "+!:uuid:x-org#card$do": {
            "/$get": [
                "=!:uuid:x-alice<#home><#email>"
            ]
        }
    },
    "(=!:uuid:x-alice/#friend)(+!:uuid:x-org#card$do$if/$true)": {
        "{$from}": {
            "/$is#friend": [
                "=!:uuid:x-alice"
            ]
        }
    },
    "(=!:uuid:x-bob)": {
        "<$uri>": {
            "&": "https://xdi.example.com/=!:uuid:x-bob/"
        }
    },
    "(=!:uuid:x-carol)": {
        "<$uri>": {
            "&": "https://xdi.example.com/=!:uuid:x-carol/"
        }
    }
}
```

12.4.2 Example Where Implied Contextual Statements are Included (Implied=1)

```
{
    "//": [
        "<$uri>",
        "=!:uuid:x-alice",
        "=!:uuid:x-bob",
        "=!:uuid:x-carol",
        "=!:uuid:x-david",
        "(=!:uuid:x-alice)",
        "(=!:uuid:x-alice/#friend)",
        "(=!:uuid:x-bob)",
        "(=!:uuid:x-carol)"
    ],
    "/$is$ref": [
        "(=!:uuid:x-alice)"
    ],
    "<$uri>": {
        "&": "https://xdi.example.com/=!:uuid:x-alice"
    },
    "=!:uuid:x-alice": {
        "//": [
            "[<#email>]",
            "<#email>",
            "<#home>",
            "<#work>",
            "#passport"
        ],
        "/#friend": [
            "=!:uuid:x-bob",
            "=!:uuid:x-carol",
            "(=!:uuid:x-alice/#friend)"
        ],
        "/#spouse": [
            "=!:uuid:x-david"
        ],
        "[<#email>]": {
            "//": [
                "<@~0>",
                "<@~1>"
            ]
        },
        "[<#email>]<@~0>": {
            "&": "alice@example.com"
        },
        "[<#email>]<@~1>": {
            "&": "asmith@example.net"
        },
        "<#email>": {
            "/$ref": [
                "=!:uuid:x-alice[<#email>]<@~0>"
            ]
        },
        "<#home>": {
            "//": [
                "<#email>]"
            ]
        },
        "<#home><#email>": {
            "/$ref": [
                "=!:uuid:x-alice[<#email>]<@~1>"
            ]
        },
        "<#work>": {
            "//": [
                "<#email>]"
            ]
        },
        "<#work><#email>": {
            "/$ref": [
                "=!:uuid:x-alice[<#email>]<@~1>"
            ]
        }
    },
    "=!:uuid:x-alice#passport": {
        "//": [
            "<#country>",
            "<#name>",
            "<#number>"
        ],
        "<#country>": {
            "&": "USA"
        },
        "<#name>": {
            "&": "Alice Smith"
        },
        "<#number>": {
            "&": "1234567"
        }
    },
    "=!:uuid:x-bob": {},
    "=!:uuid:x-carol": {},
    "=!:uuid:x-david": {},
    "(=!:uuid:x-alice)": {
        "/$ref": [
            ""
        ]
    },
    "(=!:uuid:x-alice/#friend)": {
        "//": [
            "+!:uuid:x-org#card$do",
            "+!:uuid:x-org#card$do$if",
            "(+!:uuid:x-org#card$do$if/$true)"
        ],
        "+!:uuid:x-org#card$do": {
            "/$get": [
                "=!:uuid:x-alice<#home><#email>"
            ]
        },
        "+!:uuid:x-org#card$do$if": {
            "/$true": [
                "(=!:uuid:x-alice/#friend)(+!:uuid:x-org#card$do$if/$true)"
            ]
        }
    },
    "(=!:uuid:x-alice/#friend)(+!:uuid:x-org#card$do$if/$true)": {
        "//": [
            "{$from}"
        ],
        "{$from}": {
            "/$is#friend": [
                "=!:uuid:x-alice"
            ]
        }
    },
    "(=!:uuid:x-bob)": {
        "//": [
            "<$uri>"
        ],
        "<$uri>": {
            "&": "https://xdi.example.com/=!:uuid:x-bob/"
        }
    },
    "(=!:uuid:x-carol)": {
        "//": [
            "<$uri>"
        ],
        "<$uri>": {
            "&": "https://xdi.example.com/=!:uuid:x-bob/"
        }
    }
}
```

12.5 Special Case Examples

This section provides examples of how to serialize the special cases involving solitary context nodes, object-only context nodes, and inner roots. Each is shown both with and without implied contextual statements.

12.5.1 Solitary Context Nodes

12.5.1.1 Implied=0

Statement format:

```
//=a
//=b
```

JSON format:

```
{
    "=a": {},
    "=b": {}
}
```

12.5.1.2 Implied=1

Statement format:

```
//=a
//=b
```

JSON format:

```
{
    "//": [
        "=a",
        "=b"
    ],
    "=a": {},
    "=b": {}
}
```

12.5.2 Object-Only Context Nodes

12.5.2.1 Implied=0

Statement format:

```
=a/#friend/=b
```

JSON format:

```
{
    "=a": {
        "/#friend": [
            "=b"
        ]
    }
}
```

12.5.2.2 Implied=1

Statement format:

```
//=a
//=b
=a/#friend/=b
```

JSON format:

```
{
    "//": [
        "=a",
        "=b"
    ],
    "=a": {
        "/#friend": [
            "=b"
        ]
    },
    "=b": {}
}
```

12.5.3 Both Solitary and Object-Only Context Nodes

12.5.3.1 Implied=0

Statement format:

```
=a/#friend/=b
//=c
```

JSON format:

```
{
    "=a": {
        "/#friend": [
            "=b"
        ]
    },
    "=c": {}
}
```

12.5.3.2 Implied=1

Statement format:

```
//=a
//=b
//=c
=a/#friend/=b
```

JSON format:

```
{
    "//": [
        "=a",
        "=b",
        "=c"
    ],
    "=a": {
        "/#friend": [
            "=b"
        ]
    },
    "=b": {},
    "=c": {}
}
```

12.5.4 Non-Empty Inner Root

12.5.4.1 Implied=0

Statement format:

```
(=a/#b)=x/#y/=z
```

JSON format:

```
{
    "(=a/#b)": {
        "=x": {
            "/#y": [
                "=z"
            ]
        }
    }
}
```

12.5.4.2 Implied=1

Statement format:

```
//=a
//=z
//(=a/#b)
=a/#b/(=a/#b)
(=a/#b)//=x
(=a/#b)=x/#y/=z
```

JSON format:

```
{
    "//": [
        "=a",
        "=z",
        "(=a/#b)"
    ],
    "=a": {
        "/#b": [
            "(=a/#b)"
        ]
    },
    "=z": {},
    "(=a/#b)": {
        "//": [
            "=x"
        ],
        "=x": {
            "/#y": [
                "=z"
            ]
        }
    }
}
```

12.5.5 Empty Inner Root

12.5.5.1 Implied=0

Statement format:

```
=a/#b/(=a/#b)
```

JSON format:

```
{
    "=a": {
        "/#b": [
            "(=a/#b)"
        ]
    }
}
```

12.5.5.2 Implied=1

Statement format:

```
//=a
//(=a/#b)
=a/#b/(=a/#b)
```

JSON format:

```
{
    "//": [
        "=a",
        "(=a/#b)"
    ],
    "=a": {
        "/#b": [
            "(=a/#b)"
        ]
    },
    "(=a/#b)": {}
}
```

13 XDI Addressing

The first design goal of XDI architecture is 100% addressability of all graph nodes. This is accomplished via XDI's semantic tree structure and formally defined in the ABNF rules. The patterns and rules for addressing and traversing nodes within this semantic tree structure are summarized in this section.

13.1 Semantic Tree Architecture

As summarized in the figure below from the *Introduction*, a semantic tree is a hybrid between a semantic graph and a conventional directory tree.

```
<svg>
<ellipse></ellipse>
<g>
<circle></circle>
<text>Semantic Web:</text>
<text>Description Logic</text>
<text>Knowledge Representation</text>
<text>Machine Learning</text>
<text>RDF, OWL, JSON-LD</text>
</g>
<g>
<circle></circle>
<text>Directory Tree:</text>
<text>Identity Management</text>
<text>Discovery, Access Control</text>
<text>Authentication, Authorization</text>
<text>X.500, LDAP, DNS</text>
<text>XACML, OAuth</text>
</g>
<text>Semantic Tree:</text>
<text>XDI</text>
</svg>
```

Like an RDF semantic graph, all arcs in an XDI graph are represented with subject/predicate/object triples. Like a conventional X.500 directory tree, hierarchical relationships in an XDI graph may be represented using contextual arcs that form a rooted tree structure. This combination yields the identifier and addressing features described in this section.

13.1.1 REAL Sequences

The first rule of XDI addressing within a semantic tree, enforced in the ABNF, is that the sequence of XDI identifiers composing an XDI address MUST follow the XDI REAL (Root-Entity-Attribute-Literal) sequence order. The following table lists the sequence patterns that are valid per the ABNF.

Table 21. Valid REAL Sequences

Pattern #	Root	Entity	Attribute	Literal	Example
1	x	x	x	x	`(+example-corp/#employee)=alice<#email>&`
2	x	x	x		`(+example-corp/#employee)=alice<#email>`
3	x	x			`(+example-corp/#employee)=alice`
4	x				`(+example-corp/#employee)`
5		x	x	x	`=alice<#email>&`
6		x	x		`=alice<#email>`
7		x			`=alice`
8			x	x	`$uri&`
9			x		`<$uri>`
10	x		x	x	`(+example-corp)<$uri>&`
11	x		x		`(+example-corp)<$uri>`

Note that in patterns 8, 9, 10, and 11, the attribute describes the root node, i.e., the graph itself, and not any entity in the graph. In patterns 8 and 9, the attribute describes the common root node of the current graph. In patterns 10 and 11, the attribute describes a peer root or inner root node.

Note also that a peer root node MUST appear before an inner root node in a sequence that includes both root node types.

13.1.1.1 Specialization and Generalization

Within a semantic tree, there is a standard pattern called the specialization/ generalization pattern for specializing the semantic meaning of an entity or attribute. This pattern is based entirely on the order of the XDI identifiers representing each arc in an XDI address. The rule is: each identifier in a sequence of entity identifiers specializes the final entity in the sequence, and each attribute in a sequence of attribute identifiers specializes the final attribute in the sequence.

This rule applies no matter how deep the sequence. For example, each of the following XDI addresses further specializes the final entity—in this case a bolt.

```
#bolt
#locking#bolt
#engine#locking#bolt
#airplane#engine#locking#bolt
#jet#airplane#engine#locking#bolt
+example-corp#jet#airplane#engine#locking#bolt
```

Note: although the specialization/generalization pattern happens to align with how a noun is specialized by a series of adjectives or adverbs in English, the pattern is not derived from English (or any other human language). It is a natural property of semantic trees.

In the first five cases in the example above, the specialization is one of type; each preceding entity class further specializes the type of bolt. Such type specializations may be one of two kinds:

1. **Defined specializations** are formally defined in an XDI dictionary so there are machine-understandable rules describing how the specialization either categorizes or modifies the attributes and/or relations of the specialized entity.

2. **Ad hoc specializations** do not change the semantic definition of an entity but serve only to categorize it.

In the final case in the example above, where the class sequence is put in the context of the `+example-corp` group instance, the specialization is only one of possession and does not affect the semantic definition of the entity.

The specialization/generalization pattern also applies to attributes. The following examples shows how a phone number attribute can be specialized:

```
<#phone>
<#home><#phone>
<#work><#phone>
<#fax><#phone>
<#home><#fax><#phone>
<#work><#fax><#phone>
```

In this example, none of the specializations changes the semantics of the phone number being described; they only serve to categorize it.

Note that the specialization/generalization pattern cannot be applied to peer root nodes—because they all represent peer XDI graphs—or to literal nodes—because they cannot be nested.

13.1.2 Peer Roots and XDI Discovery

Conventional directory tree architectures such as X.500, LDAP, and DNS assume a single rooted directory information tree (DIT). XDI semantic tree architecture assumes there may be any number of parallel rooted trees (peer graphs), each with its own peer root node. As discussed in the Peer Roots section, while all peer graphs have a single logical root (the XDI common root node), none of them are assumed to be authoritative for a particular branch of the graph. Authority may only be determined by consensus among a set of peers.

The process of determining this consensus is called *XDI discovery* and is defined by the XDI Discovery specification [**XDI-Discovery-V1.0**]. XDI discovery begins with the assumption that any peer root node may contain its own subgraph of other peer root nodes, including attributes such as the authoritative IRIs for their XDI endpoints. Each of these subgraphs is itself a branch of the semantic tree. An XDI discovery agent may “walk the tree” of the peer root nodes it trusts to discover the location of other peer graphs. In this process it may also determines their consensus about the authority for a particular branch of the XDI semantic tree.

The sequence of peer root nodes that need to be traversed to determine the location and authority of another peer root node is called the peer root address. Each peer root node in the address includes the XDI identifier for the entity authoritative for that peer root relative to the previous peer root. This discovery process can be applied to all XDI entities, including persons, groups, and things. Examples of peer root addresses:

```
(=example-person)
(+example-corp)(=example-person)
(+example-consortia)(+example-corp)(=example-person)
(=example-person)(*example-thing)
(=example-corp)(*example-thing)
```

Each peer graph for `(=example-person)` in the above examples represents the same natural person (because it has the same absolute XDI identifier). However each is a different peer graph because of its position in the peer root address. In the first example, the peer graph is at the common root level. This means the authority is the natural person with the following XDI entity address:

```
=example-person
```

In the second example, the peer root (=example-person) is relative to the peer root(+example-corp). This means it is authoritative for the following XDI address:

```
+example-corp=example-person
```

This is a contextual description of `=example-person` in the context of `+example-corp`, for which `+example-corp` is authoritative.

In the third example, the peer root `(+example-corp)` is itself relative to the peer root `(+example-consortia)`. This means it is authoritative for the following XDI address:

```
+example-consortia+example-corp=example-person
```

This is a contextual description of `=example-person` in the context of `+example-corp`, which in turn is a contextual description in the context of `+example-consortia`. The first entity in this sequence, `+example-consortia`, is ultimately authoritative for this entire address.

This pattern may be nested as deeply as needed in order to delegate authority for XDI peer graphs the same way authority may be delegated in DNS or other hierarchical naming and directory tree systems.

Note that because any number of peer graphs may describe the same peer root address, in XDI there is no single authoritative peer root node at which to begin discovery of that address. Instead, an XDI discovery agent may choose a specific peer root node to trust as the starting point, or it may query multiple peer root nodes and compare their answers to determine a consensus.

For more details on XDI discovery, see the XDI Discovery specification [**XDI-Discovery-V1.0**].

13.1.3 Inner Roots and XDI Reification

As described in Inner Roots in the Roots section, semantic tree architecture uses the inner graph pattern to reify a relationship so it can be further described. This pattern is fundamental to many aspects of XDI architecture including XDI messaging, XDI connections, and XDI policies.

From an XDI addressing standpoint, the inner graph pattern enables context nodes to be described and addressed in the context of either a generic or a specific relationship. For a generic relationship, the predicate of the inner graph is an XDI class, and all the root-level subjects of the inner graph are members of that class, as shown in the following example.

```
{
    "(=alice/#friend)": {
        "=bob": {
            "/#introducer": [
                "=beth"
            ],
            "<#introduction><$date>": {
                "&": "2010-11-12"
            }
        },
        "=edith": {
            "/#introducer": [
                "=beth"
            ],
            "<#introduction><$date>": {
                "&": "2010-11-13"
            }
        },
        "=frank": {
            "/#introducer": [
                "=edith"
            ],
            "<#introduction><$date>": {
                "&": "2010-11-14"
            }
        }
    }
}
```

For a specific relationship, the predicate of the inner graph is an XDI instance, and the root-level subjects of the inner graph are either attributes of that specific relationship, or entities describing the relationship, as shown in the following example.

```
{
    "(=alice/=bob)": {
        "#marriage": {
            "/#minister": [
                "=parson-brown"
            ],
            "<$date>": {
                "&": "2010-08-22"
            }
        }
    }
}
```

From the standpoint of XDI addressing, it does not matter if an inner graph is generic or specific. It both cases, the inner graph serves as the root of another complete XDI graph, and addressing within this graph works the same way as inside any XDI common graph, with one exception: an inner graph MUST NOT contain a peer root node. Only the common root node or another peer root node may contain a peer root node. However an inner root node MAY contain another inner root node.

Both generic and specific relationships described by inner graphs are the starting point for XDI link contracts. A link contract is an entity that describes the rights and permissions over one or more XDI subgraphs that an authorizing XDI authority extends to a requesting authority. Each link contract includes a policy branch that uses nested inner graphs to express the policies that must be satisfied in order to grant authorization. XDI link contracts and policy expressions are defined in the XDI Link Contracts specification [**XDI-Link-Contracts-V1.0**].

13.2 Namespace Architecture

The ABNF defines six native namespaces for XDI identifiers. Namespaces may be used with properties except in the cases in the table below, as defined in the following sections.

Dollar words are inherently immutable and absolute, and do not contain encapsulated IRIs.

Ordinals, on the other hand, are inherently relative.

Table 22. Namespace and Property Compatibility

Name-space	Mutable	Immutable	Absolute	Relative	Rooted	Nested	IRI
$	**No**	Yes*	Yes	**No**	Yes	Yes	**No**
#	**No**	Yes*	Yes	Yes	Yes	Yes	Yes
=	Yes	Yes	Yes	Yes	Yes	Yes	Yes
+	Yes	Yes	Yes	Yes	Yes	Yes	Yes
*	Yes	Yes	Yes	Yes	Yes	Yes	Yes
@	Yes	Yes	Yes**	Yes	Yes**	Yes	**No**

* Classes identifiers are immutable by definition, so they do not use the immutability symbol. See *Classes*.

** Absolute ordinal identifiers are not ordered, and only absolute ordinal identifiers may be rooted. See *Ordinals*.

Four of the namespaces also allow the use of IRIs as XDI identifiers. This allows any resource with an IRI to be described by an XDI graph. In addition, any XDI address may be transformed into a valid IRI, so all XDI graph nodes may be treated as Web resources. See the *IRIs* section..

13.2.1 Mutable and Immutable Identifiers (XDI Names and XDI Numbers)

To meet the Persistent Identification design goal, XDI namespace architecture requires the ability for XDI authorities to assign identifiers that are immutable, i.e., assigned once to identify a resource and never reassigned to identify a different resource. Immutable XDI identifiers are also called XDI numbers. The requirements for immutable identifiers are defined by the IETF URN (Uniform Resource Name) specification [**RFC 4122**].

At the same time, to meet human usability requirements, XDI namespace architecture requires the ability for XDI authorities to assign identifiers that are mutable, i.e., that may be assigned to identify one resource at one point in time and a different resource at a different point in time. Mutable XDI identifiers are also called XDI names. Examples of mutable identifiers are dynamic IP addresses, domain names that may be bought and sold by different owners, and email addresses that may be recycled to different users.

Because XDI names are typically more human-friendly than XDI numbers, they are the default form of XDI identifier. XDI addressing only requires explicit syntax to express an XDI number. All XDI identifiers except those in the class namespaces ($ and #) MUST use the ! immutability symbol if and only if the XDI authority for that identifier asserts that the identifier is immutable.

All identifiers in the XDI class namespaces are immutable by definition and MUST NOT use the immutability symbol. While an XDI class identifier MUST NOT be reassigned, its definition MAY evolve. An evolved definition SHOULD be identified with an XDI version number. See Versioning.

The XDI & literal symbol is immutable by definition and MUST NOT use the immutability symbol.

An XDI address consisting of a sequence of XDI identifiers is immutable if and only if all of the XDI identifiers in the sequence are immutable.

The name/number reference pattern defined in the Reference Relations section demonstrates how XDI graphs may use equivalence relations to map human-friendly mutable identifiers to machine-friendly immutable identifiers. Such mappings may also be cryptographically signed. Thus, while neither an XDI name nor an XDI number by itself can solve the namespace design problem known as Zooko's Triangle [zooko] https://en.wikipedia.org/wiki/Zooko%27s_triangle], the combination of an XDI name/number mapping can effectively "square Zooko's triangle".

13.2.2 Absolute and Relative Identifiers

Absolute and relative identifiers play a very different role in semantic tree architecture than in conventional directory tree or federated namespace architectures. In the latter, absolute identifiers only exist at the top level of the tree, i.e., they identify first level of nodes under the root. All other identifiers for nodes below the first level are relative to the nodes above them.

In XDI, an absolute identifier is one that identifies the same logical resource regardless of its parent context. A relative identifier is one whose scope of identification of a resource is relative to its parent context.

For example, in each of the following three XDI addresses, the identifier `=!:uuid:x-1` identifies the same natural person.

```
=!:uuid:x-1
+!:uuid:x-2=!:uuid:x-1
+!:uuid:x-3+!:uuid:x-2=!:uuid:x-1
```

In the second address, the person is represented in the context of a group that also has an absolute address. In the third address, both the person and the first group are represented in the context of a second group that also has an absolute address.

Because absolute identifiers may exist at any level of the XDI graph and provide so much semantic value, they are the default form of XDI identifier. XDI addressing only requires explicit syntax to express when an XDI identifier is relative. All XDI identifiers except those in the $ reserved class namespace MUST use the ~ relativity symbol if and only if the XDI authority for that namespace asserts that the identifier is relative.

All identifiers in the $ reserved class namespace are absolute by definition and MUST NOT use the relativity symbol. The following two XDI addresses are an example of a relative identifier for a person in the context of absolute identifiers for two different groups.

```
+!:uuid:x-4=~alice
+!:uuid:x-5=~alice
```

Unlike the previous example, no inference can be made that the person identified by `=~alice` relative to the first group has any relationship to the person identified by `=~alice` relative the second group. The equivalence of the two relative identifiers has no semantic meaning in an XDI graph. An XDI address consisting of a sequence of XDI identifiers is absolute if and only if it begins with an absolute identifier. Because of this rule, when a relative identifier is placed in the context of an absolute identifier, the combination becomes absolute. For example, in the three XDI addresses below, the combination of
`+!:uuid:x-4=~alice` represents the same unique person in all three.

```
+!:uuid:x-4=~alice
+!:uuid:x-5+!:uuid:x-4=~alice
+!:uuid:x-6+!:uuid:x-5+!:uuid:x-4=~alice
```

This rule holds true even for multiple relative identifiers. For example, in the three XDI addresses below, the combination of
`+!:uuid:x-4=~alice*~phone` represents the same unique device in all three.

```
+!:uuid:x-4=~alice*~phone
```

```
+!:uuid:x-5+!:uuid:x-4=~alice*~phone
+!:uuid:x-6+!:uuid:x-5+!:uuid:x-4=~alice*~phone
```

An XDI address consisting of a sequence of XDI identifiers that begins with a relative identifier is always relative, even if other identifiers in the sequence are absolute.

To summarize the rules, an XDI absolute identifier:

1. MUST be globally unique in the XDI logical graph.

2. MUST NOT include the relativity symbol.

3. MAY appear at any level of an XDI graph.

4. MUST be inferred as identifying the same logical resource regardless of the context in which it appears.

An XDI relative identifier:

1. MUST be unique within the scope of its parent context node.

2. MUST include the relativity symbol.

3. MUST NOT appear at the root level of an XDI graph.

4. MUST NOT be inferred as identifying the same resource relative to any other XDI context node.

For absolute identifiers, the XDI name prefix "x-" is reserved for examples and testing and SHOULD NOT be assigned for any other purpose.

13.2.3 Rooted and Nested Identifiers

For the same reason a semantic tree has different definitions of absolute and relative identifiers, it needs different terms for describing the relative position of an identifier in an XDI address.

In conventional directory tree or federated namespace architectures, an absolute identifier must be the first identifier in an address sequence because its parent is the root node. All other identifiers in the address sequence are relative.

In a semantic tree, an absolute identifier may exist at any level of the tree. So XDI uses the term rooted identifier for an identifier whose parent is a root node, and rooted address for an XDI address that begins with a rooted identifier. It uses the term nested identifier for an identifier that whose parent is not a root node, and nested address for an XDI address that begins with a nested identifier.

Per the rules in the previous section, it follows that:

1. An XDI rooted identifier MUST be an absolute identifier, and a rooted address MUST be an absolute address.

2. An XDI nested identifier MAY be either an absolute or a relative identifier, and a nested address MAY be either an absolute or a relative address.

An XDI rooted address is the equivalent of a fully-qualified address in conventional directory tree or federated namespace architectures. Those architectures assume only a single root node so there is only a single form of a rooted address. In XDI semantic tree architecture there are four forms of rooted addresses as shown in he table below.

Table 23. Rooted Address Forms

Rooted address type	Example
Common rooted	`=alice<#email>`
Peer rooted	`(+example-corp)=alice<#email>`
Inner rooted	`(+example-corp/#employee)=alice<#email>`
Peer and inner rooted	`(+example-dir)(+example-corp/#employee)=alice<#email>`

In all four examples, the rooted address is `=alice<#email>`. In the first example, the root is the common root. In the second, the root is a peer root; in the third, an inner root; and in the fourth, both a peer root and an inner root.

Note that when an XDI address begins with a peer or inner root node, that peer or inner root node is itself still rooted in the common root node. The common root node is the logical parent node of all absolute XDI addresses.

13.2.4 Public and Private Identifiers

Most directory tree and federated namespace architectures are designed for either public or private access, but not both. XDI semantic tree namespaces are equally suited for both public and private access.

Unlike the distinctions between mutable and immutable identifiers and between absolute and relative identifiers, the distinction between public and private identifiers is not syntactic, but rather one of access policy.

A *public identifier* is an XDI identifier whose XDI discovery information is available at an XDI endpoint accessible via a public link contract, i.e., a link contract that grants any requesting authority permission to access the subgraph without authentication. Public link contracts are defined in the XDI Link Contracts specification [**XDI-Link-Contracts-V1.0**].

A *private identifier* is an XDI identifier that requires a non-public link contract in order to access its XDI discovery information.

Note that the distinction between public and private identifiers in XDI does not depend on the visibility of the identifier itself. A public identifier may be kept hidden or a private identifier may be publicly known. The distinction is based on access control.

A special XDI context, `$anon`, is reserved for private identifiers whose subgraphs are intended for sharing XDI data while preserving anonymity, such as might be required for aggregated health records or quantified self data. The $anon context is defined in the XDI Privacy specification [**XDI-Privacy-V1.0**].

The ability to support both public and private identifiers—and for private identifiers to use `$rep` replacement relations (as described in Replacement Relations in the Equivalence Relations section) to protect pseudonymity is a key component of how XDI semantic tree architecture supports Privacy By Design [**pbd**].

13.3 Internationalization

To enable XDI to describe any resource that humans are able to identify in natural language, XDI identifiers are defined by the Unicode Identifier and Pattern Syntax [**UAX31**] with the addition of underscore, hyphen, and period as allowable non-initial charactes.

To enable XDI to describe any resource on the World Wide Web, XDI identifiers support the encapsulation of any IRI (Internationalized Resource Identifier) as covered in the IRI section.

13.4 Normalization and Comparison

Consistent identification of resources both within and across contexts is vital to interoperability of XDI. Therefore XDI requires users to perform any normalization of identifiers before using them as XDI identifiers. The XDI endpoint itself MUST NOT attempt to perform any normalization or folding. We cover some specific cases below.

The following normalization rules apply to all XDI identifiers except encapsulated IRIs, which are covered in the following section.

These normalization rules do not by themselves prevent homographic attacks (spoofing of an XDI identifier by using look-alike characters from a different script). XDI authorities—and in particular those who act as XDI registries—SHOULD impose identifier registration policies that prevent homographic attacks.

13.4.1 Upper and Lower Case

XDI processors MUST NOT do case folding or case-insensitive comparison of XDI identifiers. Client systems which internally use case-insensitive identifiers (e.g. URNs; Macintosh filenames; SQL) are expected to case-fold names to a consistent normalized format (which SHOULD be all lower case, for compatibility between XDI applications) before introducing it to an XDI graph.

However, for percent-encoded bytes in `%XX` format, the two hex digits SHOULD be upper case `ABCDEF`, not lower case `abcdef`.

13.4.2 Unicode Normalization Forms

XDI identifiers SHOULD be in Unicode Normalization Form NFKC. [**UAX15**] Most existing Unicode text meets this criterion.

13.5 IRIs

To be fully compliant with W3C World Wide Web architecture, XDI addressing incorporates IRIs in two ways:

1. Any World Wide Web resource identified by an IRI MAY be described in an XDI graph by encapsulating the IRI as an XDI identifier.
2. Any XDI graph node identified by an XDI address MAY be expressed as a World Wide Web resource by appending the XDI address to an IRI that identifies the host XDI graph.

This section defines the rules in both directions.

13.5.1 Describing IRIs in XDI Addresses

Describing an IRI-identified resource in XDI requires two steps: Normalize the IRI (to prevent misinterpretation of where the IRI terminates when encapsulated within an XDI identifier). Encapsulate the IRI within an XDI identifier by enclosing it in parentheses as required by the XDI ABNF. Examples:

Table 24. IRI Encapsulation

IRI	XDI address encapsulating IRI
`http://example.com/`	`+(http://example.com/)`
`mailto:alice@example.com`	`=(mailto:alice@example.com)`
`https://example.com/item#id`	`*!(https://example.com/item#id)`

To normalize an IRI prior to encapsulation, the following steps MUST be performed in order and exactly once (since they are not idempotent).

1. Put the IRI into its absolute normalized form, including any required percent-encoding, as required by the specification for the applicable IRI scheme and by RFC 3987 [**RFC 3987**].

2. Percent-encode all percent % characters as %25.

3. Percent-encode all right parentheses) characters as %29.

If it is later necessary to extract the original IRI, the following steps MUST be performed on the encapsulated IRI in order and exactly once (since they are not idempotent).

1. Decode all %29 percent-encoding as right parentheses) characters.

2. Decode all %25 percent-encoding as percent % characters.

Note that when the IRI is encapsulated as an XDI identifier, semantic meaning is added—at a minimum, a prefixed XDI context symbol (and potentially an immutability symbol and/or a relativity symbol). So there is always a difference between the semantic meaning of the XDI identifier and the original IRI.

In addition, although the resource identified by the XDI identifier MUST be the same resource identified by the IRI on the World Wide Web, the XDI graph node representing the resource in the logical XDI graph MUST NOT be the same resource identified by the IRI on the World Wide Web. This separation between the Web resource and the XDI description of the Web resource avoids the HTTPRange-14 issue [**httprange14**].

13.5.2 Transforming XDI Addresses into IRIs

Transforming an XDI address identifying an XDI graph node into a valid IRI identifying a World Wide Web resource address requires two steps:

1. Normalize the XDI address (to turn it into a valid relative IRI and prevent misinterpretation by an IRI parser).

2. Append the XDI address as a relative IRI to a base IRI identifying the host XDI endpoint.

To normalize the XDI address, the following steps MUST be performed in order and exactly once (since they are not idempotent).

1. Percent-encode all percent % characters as %25.

2. Percent-encode each of the characters in the table below.

Table 25. Percent-encoding required

Character name	Character	Encoded
Dollar	$	%24
Hash	#	%23
Equals	=	%3D
Plus	+	%2B
Star, asterisk	*	%40
At sign	@	%26
Ampersand	&	%7C
Pipe, vertical bar	|	%3C
Chevrons, angle brackets	< >	%3C %3E
[Square] brackets	[ ]	%5B %5D
Braces, curly brackets	{ }	%7B %7D

If it is later necessary to restore the original XDI address, the following steps MUST be performed on the normalized XDI address in order and exactly once (since they are not idempotent).

1. Decode all the percent-encoded characters in the table above.

2. Decode all `%25` percent-encoding as percent `%` characters.

Once the XDI address is transformed into a relative IRI, it may be appended to a base URI that identifies the host XDI endpoint as defined in section 6.5 of RFC 3987 [**RFC 3987**]. It is RECOMMENDED that the base URI end with a forward slash. To the extent that the XDI authority is also the authority for the DNS name, it is RECOMMENDED that DNS host name of the IRI for an XDI endpoint begin with `xdi`.

If an XDI address does not contain percent-encoding, the resulting IRI will only reflect percent-encoding of the XDI syntax characters. For example, start with the following XDI address:

```
=alice<#email>
```

If the base IRI for the XDI endpoint hosting this XDI address is:

```
http://xdi.example.com/
```

Then the resulting fully qualified IRI would be:

```
http://xdi.example.com/%3Dalice%3C%23email%3E
```

If the XDI address contains percent-encoding, the resulting IRI will have percent-encoding of both the original percent-encoding as well of the XDI syntax characters. For example, take this XDI address:

```
=alice*some%20pet%20name
```

When normalized and appended to the same base URI above, the fully qualified IRI would be:

```
http://xdi.example.com/%3Dalice%2Asome%2520pet%2520name
```

13.6 XDI Schemes

In addition to context, immutability, and relativity symbols, in some cases XDI identifiers require additional syntactic structure in order to express identifiers with specific properties. To meet this requirement, XDI addressing syntax uses a mechanism analogous to URIs and IRIs, xalled XDI schemes.

With URIs and IRIs, an absolute identifier must begin with a scheme name followed by a colon (e.g., “`http:`”, “`ftp:`”, “`mailto:`”). Scheme names must also use a limited character set.

With XDI identifiers, a scheme is an optional syntactic feature. If an XDI identifier includes an XDI scheme, the following rules apply:

1. An XDI scheme name MUST begin and end with a colon character and include a sequence of one or more characters as allowed by the `xdi-scheme` rule in the XDI ABNF.

2. The scheme name MUST follow the XDI context symbol and, if present, the immutability symbol and/or relativity symbol.

3. The identifier following the scheme name MUST be valid according to the scheme specification.

As with URIs and IRIs, XDI schemes are extensible. However unlike XDI dictionary spaces, which may be specialized by any XDI authority in its own namespace, the XDI scheme namespace is shared across all XDI authorities. To prevent collisions, it is RECOMMENDED that:

1. Members of an XDI community requiring a new XDI scheme collaborate to develop a scheme that does not conflict with existing XDI schemes.
2. A new scheme be documented in a public specification that includes the scheme name, the purpose of the scheme, ABNF rules for validating identifiers conformant to the scheme, and any security, privacy, or other special considerations for using the scheme.
3. New scheme specifications be registered with the OASIS XDI Technical Committee.

Note that use of an XDI scheme does not alter the requirement for the XDI identifier to be unique in the scope of its parent context—and for all XDI absolute identifiers to be globally unique. Indeed, the motivation for the two XDI schemes defined in the following sections is to establish interoperable standards for how XDI identifiers can meet this global uniqueness requirement.

13.6.1 UUID (Universally Unique Identifier)

UUIDs as defined by RFC 4122 [**RFC 4122**] are widely used in distributed computing as globally unique identifiers that do not require a central registration authority. When an XDI authority needs to generate an absolute XDI identifier that does not have any other specific properties (such as a cryptographic identifier—see the next section), the use of the XDI UUID scheme is RECOMMENDED.

As defined in the XDI ABNF, the XDI UUID scheme name is `:uuid:`. An XDI UUID MUST be a valid UUID conforming to RFC 4122. It SHOULD be a Version 4 UUID as specified in sections 4.1.3 and 4.4 of RFC 4122. Implementers SHOULD follow the recommendations in RFC 4122 and RFC 1750 for generating random numbers with sufficient entropy.

Although the probability of collision of two UUIDs is extremely small, XDI authorities SHOULD always check to ensure the uniqueness of an XDI identifier within its parent context. This is particular important for XDI authorities who offer XDI registry and discovery services to other XDI authorities.

13.6.2 CID (Cryptographic Identifier)

The XDI CID scheme family is reserved for identifiers with cryptographic properties. As defined in the XDI ABNF, the XDI CID scheme prefix is `:cid-:`. The hyphen MUST be followed by one or more digits identifying a specific XDI CID scheme. Specific XDI CID schemes will be defined in either the XDI Scheme specification or the XDI Cryptography specification.

14 Versioning

Using semantic tree architecture, XDI graphs can model versioning uniformly and interoperably at all levels of the tree. The overall rules for XDI versioning are defined in the XDI Versioning specification. The minimal rules for expressing the version of an XDI graph conformant with this specification are defined here.

The XDI reserved class name for versions is `$v`. Since a version tree is by definition an ordered collection of version instances, a version class will always appears as a collection:

1. `([$v])` for a graph root version collection.
2. `[$v]` for an entity version collection.
3. `[<$v>]` for an attribute version collection.

The members of the collection must be ordinal identifiers expressing a ordered set of version instances.

To express that it conforms to a specific version of this specification, an XDI graph MAY include an XDI version statement. An XDI version statement is an XDI type statement that uses an XDI version collection to describe the common root node of the XDI graph.

The current specification is the first version of the XDI Core specification, so its version instance identifier is `(@~0)`. Therefore if an XDI graph needs to assert conformance with this version of this specification, it MUST include the following XDI version statement:

```
{
    "/$is#": [
        "($xdi)([$v])(@~0)"
    ]
}
```

It is NOT REQUIRED for an XDI graph to contain an XDI version statement.

15 Appendix A: Collected ABNF

```
xdi-graph               = *( xdi-statement / CRLF )
xdi-statement           = contextual-statement / literal-statement / relational-statement

contextual-statement    = direct-contextual / inverse-contextual

direct-contextual       = peer-root-direct / inner-root-direct / entity-direct / attr-dire
peer-root-direct        = *peer-root "//" peer-root
inner-root-direct       = root-address "//" inner-root
entity-direct           = entity-address "//" entity
attr-direct             = attr-address "//" attr

inverse-contextual      = peer-root-inverse / inner-root-inverse / entity-inverse / attr-i
peer-root-inverse       = peer-root   "/$is()/" *peer-root
inner-root-inverse      = inner-root   "/$is()/" root-address
entity-inverse          = entity "/$is()/" entity-address
attr-inverse            = attr   "/$is()/" attr-address

literal-statement       = entity-address 1*attr "/&/" value
literal-var-statement   = entity-address 1*attr "/{&}/" value-variable
value-variable          = "{" attr-class "}"

relational-statement    = direct-relational / inverse-relational / relation-definition
direct-relational       = xdi-address "/" 1*entity "/" xdi-address
inverse-relational      = xdi-address "/$is" 1*entity "/" xdi-address

relation-definition     = direct-domain / inverse-domain / direct-range / inverse-range

direct-domain           = direct-entity-domain / direct-attr-domain
direct-entity-domain    = root-address 1*definition "/(/)/" root-address 1*definition
direct-attr-domain      = root-address *definition 1*attr-definition "/(/)/" root-address

inverse-domain          = inverse-entity-domain / inverse-attr-domain
inverse-entity-domain   = root-address 1*definition "/$is(/)/" root-address 1*definition
inverse-attr-domain     = root-address 1*definition "/$is(/)/" root-address *definition 1*

direct-range            = direct-entity-range / direct-attr-range
direct-entity-range     = root-address 1*definition "/(/)#/" root-address 1*definition
direct-attr-range       = root-address 1*definition "/(/)#/" root-address *definition 1*at

inverse-range           = inverse-entity-range / inverse-attr-range
inverse-entity-range    = root-address 1*definition "/$is(/)#/" root-address 1*definition
inverse-attr-range      = root-address *definition 1*attr-definition "/$is(/)#/" root-addr

xdi-address             = root-address / entity-address / attr-address / literal-address
root-address            = *peer-root *inner-root
entity-address          = root-address *entity
attr-address            = entity-address *attr
literal-address         = entity-address 1*attr "&"
```

```
peer-root               = peer-root-instance / peer-root-variable
peer-root-instance      = "(" entity ")"
peer-root-variable      = "{" peer-root-instance "}"

inner-root              = inner-root-instance / inner-root-variable
inner-root-instance     = inner-root-peer-root / inner-root-entity
inner-root-peer-root    = "(" *peer-root "/" *entity ")"
inner-root-entity       = "(" *entity "/" *entity ")"
inner-root-variable     = "{" inner-root-instance "}"

entity                  = singleton / collection / definition / variable / meta-variable
singleton               = instance / class
collection              = "[" class "]"
definition              = "|" ( singleton / collection ) "|"
variable                = "{" ( singleton / collection / definition ) "}"
meta-variable           = "{" variable "}"

instance                =  person / group / thing / ordinal
person                  = "=" [ "!" ] [ "~" ] id-string
group                   = "+" [ "!" ] [ "~" ] id-string
thing                   = "*" [ "!" ] [ "~" ] id-string
ordinal                 = "@" [ "!" ] [ "~" ] ordinal-string

class                   = reserved-class / unreserved-class / "$" / "#" / "=" / "+" / "*"
reserved-class          = "$" xdi-name
unreserved-class        = "#" [ "~" ] id-string

attr                    = attr-singleton / attr-collection / attr-definition / attr-variab
attr-singleton          = attr-class / attr-instance
attr-collection         = "[" attr-class "]"
attr-definition         = "|" ( attr-singleton / attr-collection ) "|"
attr-variable           = "{" ( attr-singleton / attr-collection / attr-definition ) "}"
attr-meta-variable      = "{" attr-variable "}"
attr-class              = "<" class ">"
attr-instance           = "<" instance ">"

id-string               = xdi-name / xdi-scheme / encap-iri
ordinal-string          = int / other-scheme

xdi-name                = ID_Start *( ID_Continue / "_" / "-" / "." )
pct-encoded             = "%" HEXDIG HEXDIG

xdi-scheme              = uuid-scheme / cid-scheme / other-scheme
uuid-scheme             = ":uuid:" 8HEXDIG "-" 4HEXDIG "-" 4HEXDIG "-" 2HEXDIG 2HEXDIG "-"
cid-scheme              = ":cid-" 1*DIGIT ":" xdi-name
other-scheme            = ":" ( lower-alpha / DIGIT ) *( lower-alpha / DIGIT / "_" / "-" /

encap-iri               = "(" absolute-iri ")"
absolute-iri            = iri-scheme ":" 1*iri-char
iri-scheme              = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )
iri-char                = safe-char / %xA0-EFFFD / pct-encoded
safe-char               = unreserved / reserved / gen-delims / safe-sub-delims
unreserved              = ALPHA / DIGIT / "-" / "." / "_" / "~"
reserved                = gen-delims / safe-sub-delims
```

```
gen-delims              = ":" / "/" / "?" / "#" / "[" / "]" / "@"
safe-sub-delims         = "!" / "$" / "&" / "(" / "*" / "+" / "," / ";" / "="

value                   = "false" / "null" / "true" / object / array / number / string
object                  = begin-object [ member *( value-separator member ) ] end-object
member                  = string name-separator value
array                   = begin-array [ value *( value-separator value ) ] end-array

begin-array             = ws %x5B ws  ; [ left square bracket
begin-object            = ws %x7B ws  ; { left curly bracket
end-array               = ws %x5D ws  ; ] right square bracket
end-object              = ws %x7D ws  ; } right curly bracket
name-separator          = ws %x3A ws  ; : colon
value-separator         = ws %x2C ws  ; , comma

number                  = [ "-" ] int [ frac ] [ exp ]
exp                     = [ "e" / "E" ] [ "-" / "+" ] 1*DIGIT
frac                    = "." 1*DIGIT
int                     = "0" / ( %x31-39 *DIGIT )   ; no leading zeros

string                  = quotation-mark *char quotation-mark

char                    = unescaped / backslash ( quotation-mark / backslash /
                          "/" / "b" / "f" / "n" / "r" / "t" / "u" 4HEXDIG )

backslash               = %x5C           ; \ reverse solidus U+005C
quotation-mark          = %x22           ; " quotation mark  U+0022

unescaped               = %x20-21 / %x23-5B / %x5D-10FFFF

ws                      = *( %x20 / %x09 / %x0A /  %x0D )

ALPHA                   = %x41-5A / %x61-7A                            ; A-Z, a-z
DIGIT                   = %x30-39                                      ; 0-9
HEXDIG                  = %x30-39 / %x41-46 / %x61-66                  ; 0-9, a-f, A-F
CRLF                    = %x0D / %x0A / ( %x0D %x0A )
```

16 Appendix B: Acknowledgments

In addition to the Editors, the XDI Technical Committee gratefully acknoledges the contributions of its active members:

- Christopher Allen
- Daniel Blum
- Les Chasen
- Peter Davis
- Dr. Phillip Windley
- Dr. Lionel Wolberger
- Ning Zhang

The Committee would also like to thank previous members:

- Geoffrey Strongin (Past Co-Chair)
- Bill Barnhill (Past Co-Chair)
- Andy Dale
- Victor Grey
- Jim Fournier
- Mike Schwartz
- John Bradley
- Aldo Castaneda
- Kaliya (IdentityWoman)
- William Dyson

Finaly, the Committee would like to thank OASIS staff members for their ongoing support: Chet Ensign, Paul Knight, Jamie Clark, Dee Schur, Robin Cover, and Scott McGrath.

17 Appendix C: Revision History

Table 26. Table

Revision	Date	Changes Made			
CSD 01	29 October 2015	First Version			