FIX TRADING COMMUNITY™
INDUSTRY-DRIVEN • INDEPENDENT • NEUTRAL

FIX Orchestra Technical Specification

Release Candidate 3

THIS DOCUMENT IS A RELEASE CANDIDATE FOR A PROPOSED FIX TECHNICAL STANDARD. A RELEASE CANDIDATE HAS BEEN APPROVED BY THE GLOBAL TECHNICAL COMMITTEE AS AN INITIAL STEP IN CREATING A NEW FIX TECHNICAL STANDARD. POTENTIAL ADOPTERS ARE STRONGLY ENCOURAGED TO BEGIN WORKING WITH THE RELEASE CANDIDATE AND TO PROVIDE FEEDBACK TO THE GLOBAL TECHNICAL COMMITTEE AND THE WORKING GROUP THAT SUBMITTED THE PROPOSAL. THE FEEDBACK TO THE RELEASE CANDIDATE WILL DETERMINE IF ANOTHER REVISION AND RELEASE CANDIDATE IS NECESSARY OR IF THE RELEASE CANDIDATE CAN BE PROMOTED TO BECOME A FIX TECHNICAL STANDARD DRAFT.

DISCLAIMER

THE INFORMATION CONTAINED HEREIN AND THE FINANCIAL INFORMATION EXCHANGE PROTOCOL (COLLECTIVELY, THE "FIX PROTOCOL") ARE PROVIDED "AS IS" AND NO PERSON OR ENTITY ASSOCIATED WITH THE FIX PROTOCOL MAKES ANY REPRESENTATION OR WARRANTY, EXPRESS OR IMPLIED, AS TO THE FIX PROTOCOL (OR THE RESULTS TO BE OBTAINED BY THE USE THEREOF) OR ANY OTHER MATTER AND EACH SUCH PERSON AND ENTITY SPECIFICALLY DISCLAIMS ANY WARRANTY OF ORIGINALITY, ACCURACY, COMPLETENESS, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. SUCH PERSONS AND ENTITIES DO NOT WARRANT THAT THE FIX PROTOCOL WILL CONFORM TO ANY DESCRIPTION THEREOF OR BE FREE OF ERRORS. THE ENTIRE RISK OF ANY USE OF THE FIX PROTOCOL IS ASSUMED BY THE USER.

NO PERSON OR ENTITY ASSOCIATED WITH THE FIX PROTOCOL SHALL HAVE ANY LIABILITY FOR DAMAGES OF ANY KIND ARISING IN ANY MANNER OUT OF OR IN CONNECTION WITH ANY USER'S USE OF (OR ANY INABILITY TO USE) THE FIX PROTOCOL, WHETHER DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL (INCLUDING, WITHOUT LIMITATION, LOSS OF DATA, LOSS OF USE, CLAIMS OF THIRD PARTIES OR LOST PROFITS OR REVENUES OR OTHER ECONOMIC LOSS), WHETHER IN TORT (INCLUDING NEGLIGENCE AND STRICT LIABILITY), CONTRACT OR OTHERWISE, WHETHER OR NOT ANY SUCH PERSON OR ENTITY HAS BEEN ADVISED OF, OR OTHERWISE MIGHT HAVE ANTICIPATED THE POSSIBILITY OF, SUCH DAMAGES.

DRAFT OR NOT RATIFIED PROPOSALS (REFER TO PROPOSAL STATUS AND/OR SUBMISSION STATUS ON COVER PAGE) ARE PROVIDED "AS IS" TO INTERESTED PARTIES FOR DISCUSSION ONLY. PARTIES THAT CHOOSE TO IMPLEMENT THIS DRAFT PROPOSAL DO SO AT THEIR OWN RISK. IT IS A DRAFT DOCUMENT AND MAY BE UPDATED, REPLACED, OR MADE OBSOLETE BY OTHER DOCUMENTS AT ANY TIME. THE FIX GLOBAL TECHNICAL COMMITTEE WILL NOT ALLOW EARLY IMPLEMENTATION TO CONSTRAIN ITS ABILITY TO MAKE CHANGES TO THIS SPECIFICATION PRIOR TO FINAL RELEASE. IT IS INAPPROPRIATE TO USE FIX WORKING DRAFTS AS REFERENCE MATERIAL OR TO CITE THEM AS OTHER THAN “WORKS IN PROGRESS”. THE FIX GLOBAL TECHNICAL COMMITTEE WILL ISSUE, UPON COMPLETION OF REVIEW AND RATIFICATION, AN OFFICIAL STATUS ("APPROVED") OF/FOR THE PROPOSAL AND A RELEASE NUMBER.

No proprietary or ownership interest of any kind is granted with respect to the FIX Protocol (or any rights therein).

Document History

Revision	Date	Author	Revision comments
Release Candidate 3	March 20, 2018	Don Mendelson Silver Flash LLC	Initial draft
	March 21, 2018	Don Mendelson Silver Flash LLC	Added glossary terms

Table of Contents

1 Introduction

1.1 Objectives

FIX Orchestra was conceived as **machine readable rules of engagement** between counterparties. As such, it is a standard for exchange of metadata about the behavior of FIX applications. Orchestra is intended to cut time to onboard counterparties.

The contents of Orchestra files are machine readable (that is, processed as data) and may include:

- Message structure by each scenario, implemented as an extension of FIX Repository.
- Accepted values of enumerations by message scenario
- Workflow: when I send this message type under this condition, what can I expect back?
- How external states affect messages, e.g. market phases

- Express a condition such as for a conditionally required field using a Domain Specific Language (DSL)
- Document and exchange the Algorithmic Trading Definition Language (FIXatdl) files associated with a FIX service offering
- FIX session identification and transport configuration

From the contents, firms and vendors will be enabled to develop tools to automate configuration of FIX engines and applications, and generation of code, test cases, and documentation. The various aspects are not an all-or-nothing proposition, however. Users may implement only the features that they find most beneficial, and add features as needed.

Orchestra supports but does not change FIX protocol itself in any way, nor does it obsolete existing FIX engines or tools.

1.2 Design principles

As a standard for delivering metadata about FIX messages and application behavior, Orchestra relies on technologies that are well supported across all popular platforms and programming languages, particularly XML and XML Schema.

Since Orchestra is primarily used at design time or compile time rather than run time, high performance characteristics such as low latency are not a major concern.

In future, Orchestra may be ported to alternative technologies, particularly Web Ontology Language (OWL). However, the primary XML technologies will continue to be supported for the foreseeable future.

1.3 Glossary

Actor—either a counterparty to a FIX session or an external entity that holds state relevant to application or session behavior. An actor can take actions such as assigning state or transitioning a state machine, and it can send messages.

Code set-- A finite set of the valid values of a data element. Each unique valid value is called a code.

Datatype – the value space of a data element, possibly including enumerated values, precision or range. Some types have additional attributes, e.g. epoch and time zone of a date. Value space is at the application layer (layer 7 of the OSI model).

Discriminator—a field that modifies the value space of another field. The combination of the value field and its discriminator is variously called a choice, discriminated union, tagged union, or variant.

Encoding – a wire format for data representation. Also known as lexical space or the presentation layer (layer 6) in a protocol stack.

Extension Pack (EP) – an interim publication between major versions of a FIX standard.

L**exical space** – the representation of a data element. It belongs to the presentation layer. For character based encodings, it is defined as a particular sequence of characters. For binary encodings, it may involve mapping to primitive data types supported by computing platforms.

Pedigree – recorded history of an artifact

Provenance – a record of ownership of an artifact

Scenario—a use case of a message type

Semantic—pertaining to the meaning of a message element, regardless of its representation.

Session protocol – a protocol concerned with the reliable delivery of messages over a transport. Layer 5 in the OSI protocol model.

State machine – A behavior model that has finite, discrete values called states and defined transitions between states.

Tag – a unique numeric identifier of a message element, especially a field identifier.

Value space – the type of a data element and its possible range of values. Value space belongs to the application layer (semantics) and should be independent of encoding (presentation layer) and programming language.

XML schema–defines the elements and attributes that may appear in an XML document. The Orchestra schema is defined in W3C (XSD) schema language since it is the most widely adopted format for XML schemas.

1.4 Documentation

1.4.1 Specification terms

These key words in this document are to be interpreted as described in Internet Engineering Task Force RFC2119. These terms indicate an absolute requirement for implementations of the standard: "**must**", or "**required**".

This term indicates an absolute prohibition: "**must not**".

These terms indicate that a feature is allowed by the standard but not required: "**may**", "**optional**". An implementation that does not provide an optional feature must be prepared to interoperate with one that does.

These terms give guidance, recommendation or best practices: "**should**" or "**recommended**". A recommended choice among alternatives is described as "**preferred**".

These terms give guidance that a practice is not recommended: "**should not**" or "**not recommended**".

1.4.2 Document format

In this document, these formats are used for technical specifications and data examples.

XML element and attribute names as well as FIX field and message names appear in this font: `codeSet`

This is a sample XML snippet:

```
<fixr:field id="59" name="TimeInForce" type="TimeInForceCodeSet"/>
```

1.5 References

1.5.1 Related FIX Standards

For FIX semantics, see the current FIX message specification, which is currently FIX 5.0 Service Pack 2 with Extension Packs.

1.5.2 Dependencies on other standards

Orchestra imports Dublin Core XML schemas version 2008-02-11 for artifact provenance. Dublin Core is standardized as IETF RFC 5013 and ISO 15836.

XML 1.1 schema standards are located here W3C XML Schema

Incremental changes to an XML file may be represented by the format described in IETF RFC 5261.

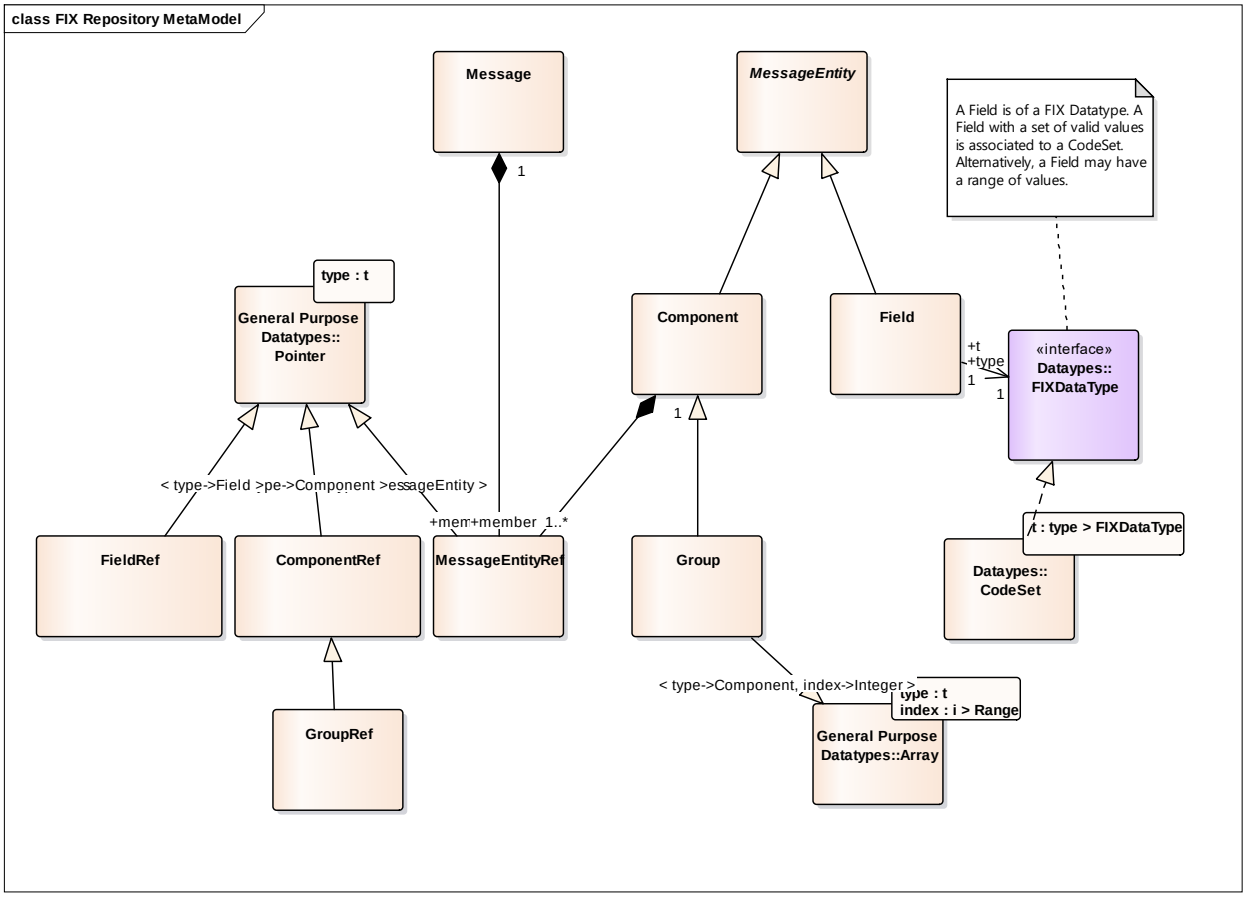
2 Metamodel

One conceptual metamodel governs all representations of FIX Orchestra and Repository knowledge bases. The XML schema and any other representations that may be developed in future, such as semantic ontologies, should be considered implementations of this common metamodel.

The metamodel presented does not strictly conform to the UML Meta-Object Facility architecture.

2.1 Message structures

The UML metamodel depicted below is a conceptual view of message structures.



2.1.1 Message structure abstractions

Field – carries a specific business meaning (semantics) as described in FIX specifications or other protocol. A pointer to a field is a **fieldRef**. The data domain of a field is either a datatype or a code set.

Datatype – the value space of a class of fields. FIX has about 20 datatypes.

Code set – a set of valid values of a field. They must all be of the same datatype.

C**omponent** – a sequence of fields and nested components. There are two types of components, common block and repeating group. A common block is a component designed to specified once in detail but reused in multiple message types by reference. A pointer to a component is **componentRef**.

Group, or repeating group – like a common block but with one additional feature: it represents an *array of* blocks to be sent on the wire. A pointer to a group is **groupRef**.

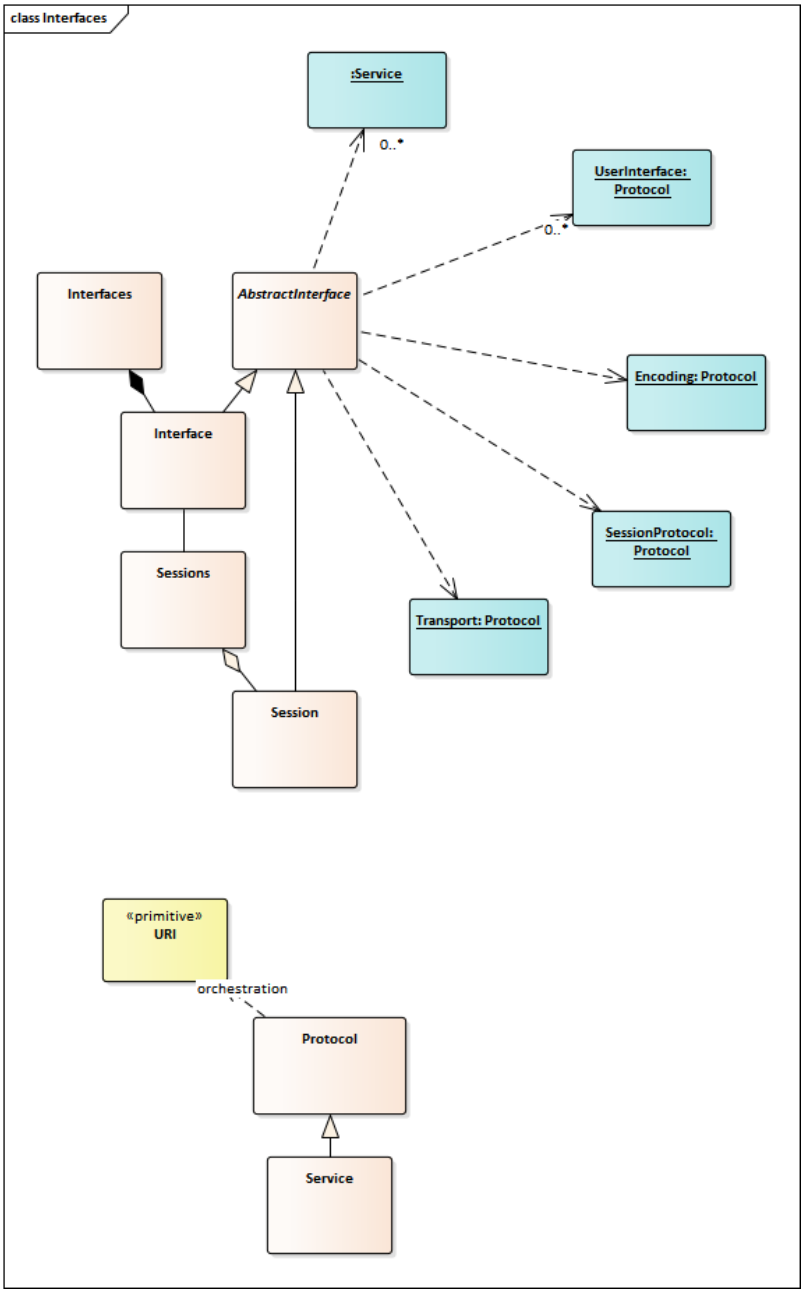
Message – a unit of information sent on the wire between counterparties. A message is composed of components and fields. A pointer to a message is a **messageRef**.

2.1.2 General Purpose Datatypes

Pointer and array abstractions are defined by standard ISO 11404. The code set abstraction is described in that standard as “state” type.

2.2 Interfaces

The interface metamodel is an abstraction of a service offerings and session provisioning. This UML model depicts the main classes.



2.2.1 Interface abstractions

Protocol—a standard for communications. The Open Systems Interconnection model (OSI) defines protocols as a layered stack, including application layer and user interface at the top, presentation layer (encoding), session layer, and transport layer. Each protocol depends on lower layers for services. Layers below transport layer are out of scope for Orchestra.

Any message-oriented protocol may have an orchestration attribute that consists of a URI. It is a link to an Orchestra file that describes message structures and workflow.

Service—a service offering by a counterparty. A service is an application layer protocol.

Interface—a collection of protocols and services exposed by a counterparty. A counterparty may offer more than one interface for different purposes. An interface may be configured for one or more service offerings and all the protocols that make up a communication stack. Also, an interface may contain any number of session configurations.

Session—a specific usage of an interface. A session has one or more identifiers. It inherits services and protocols from its parent interface, but it may have further refinement or overrides of protocol settings, such as a transport address.

3 Orchestra and Repository XML Schema

3.1 XML Schema (XSD)

FIX Orchestra and Repository 2016 Edition share a common XML schema. The two forms are only distinguished by usage. If a file only contains message structures and message documentation, it may be referred to as a Repository file. If it additionally contains work flow, state variables, conditional logic and so forth, then it is called an Orchestra file. In other words, Orchestra is a superset of Repository features.

3.1.1 Conformance

All published Repository and Orchestra files **must** conform to the standard XML schema. This can be validated with common XML parsers and related tools.

3.1.2 Schema location

The XML schema is currently available in GitHub project fix-orchestra module repository2016. Upon promotion to draft standard, it will be made available at a URI consistent with its XML namespace.

3.1.3 Root element

The root element an Orchestra XML file is <repository>. An Orchestra repository file contains all the message structures and workflow elements pertaining to a single protocol version. If an organization supports multiple versions of FIX, it should supply an Orchestra file for each.

The name attribute of <repository> identifies an implementation of a protocol. The name should remain stable over minor revisions. The version attribute should, on the other hand, should be unique for any substantive change to the protocol. If the hasComponents attribute is true, messages in the protocol are composed from components (see below); else they are flat definitions only composed of fields.

This snippet shows that element with required namespaces and attributes:

```
<fixr:repository xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 xmlns:functx="http://www.functx.com"
                 xmlns:fixr="http://fixprotocol.io/2016/fixrepository"
                 xmlns:dc="http://purl.org/dc/elements/1.1/"
                 name="FIX.5.0SP2"
                 version="FIX.5.0SP2_EP240"
                 hasComponents="1">
```

3.1.4 Supplementary documentation

See the separate document “FixRepository2016.html” for a detailed technical reference for the Orchestra and Repository XML schema. The remainder of this section serves as an overview and explains motivations for the design.

3.1.5 Protocol relationship

The schema was primarily designed to describe metadata about FIX protocols. However, it was also intended to be generic enough to work with other common financial industry protocols, especially when FIX is used in combination with other protocols, or a translation must be performed between protocols.

Usage should be supported for all phases of financial industry workflows, including pre-trade, trade, and post-trade flows.

3.2 Content ownership and history

3.2.1 Provenance

The `<metadata>` element is used to identify a particular Orchestra file and the issuer of that file. It can contain any of the elements defined by the Dublin Core XML schema. Recommended elements include publisher, date, and rights.

3.2.2 Pedigree

Most message elements in the schema support a complete history of creation, change and deprecation with support of attribute group `entityAttribGrp`. Each historical event is qualified by its protocol version and optionally, extension pack (EP), an interim publication between major versions.

Code element with pedigree

```
<code value="3" name="LocalCommission" added="FIX.4.0"
deprecated="FIX.5.0SP2" deprecatedEP="204"/>
```

3.3 Features for document and FIXML generation

The XML schema retains features that have long been used to generate FIX documentation and other outputs. These elements are optional.

3.3.1 Abbreviations

The `<abbreviations>` element tree contains approved abbreviations. One use is to shorten element names in FIXML schema generation. The schema enforces uniqueness of `<abbreviation>` elements by their `name` attribute. The explanation of an `<abbreviation>` element is contained by child `<annotation>/<documentation>` elements; see metadata below.

3.3.2 Categories

The `<categories>` element tree is used to associate FIX elements to business classifications, such as order handling, market data, and so forth, for documentation generation. Also, categories are used to organize FIXML schema files.

3.3.3 Sections

The `<sections>` element tree names document volumes. Traditionally, they have been organized around pre-trade, trade, and post-trade information flows.

3.3.4 Metadata about any element

The schema provides features to provide metadata about almost any element. All such metadata appears under element `<annotation>`. There is no limit to the number of metadata entries per `<annotation>` element.

3.3.4.1 Documentation

A `<documentation>` element can carry any description of its ancestor element. The content (text node) may be of any format, such as XHTML, markdown, or HTML5. The XML parser is instructed not to

validate the free-form content. Tools such as XSLT may be used to extract documentation from an Orchestra file and compile external documents.
Multiple languages can be supported by specifying the language of each element in its `langId` attribute. Also, multiple categories of documentation are supported by populating the `purpose` attribute. Suggested values of `purpose` include "SYNOPSIS", "ELABORATION", "EXAMPLE", and "DISPLAY".

Field element with documentation

```
<fixr:field id="45" name="RefSeqNum">
   <fixr:annotation>
   <fixr:documentation langId="en-us" purpose="SYNOPSIS">Reference
message sequence number</fixr:documentation>
   </fixr:annotation>
</fixr:field>
```

3.3.4.2 Appinfo

The `<appinfo>` element is similar to `<documentation>` in that it can support multiple languages and multiple purposes. It has an additional attribute, `specURL`, to cross-reference external documentation.

3.3.5 Rendering hints

The optional attribute `rendering` may be used to suggest how a message or element should be generated or rendered in a user interface. The value of the attribute is free-form and is not validated by the Orchestra schema.

3.4 Unique identifiers

Practically all elements in the XML schema have a `name` attribute, a numeric `id` attribute or both. These values must be unique within their respective element types within a given Orchestra file. To avoid collisions, names and IDs of deprecated elements should never be reused.

Additionally, all message elements have a provision for a globally unique, persistent object identifier in the `oid` attribute. The `oid` values will be assigned in accordance with a protocol to prevent collisions between Orchestra users. The XML schema allows either a URI or dotted notation for the attribute. The attribute is optional. If used, the OID must never change when an Orchestra is updated, and the same identifier must be carried forward to all representations of the same element, such as XML and OWL.

3.5 Datatypes

3.5.1 FIX datatypes

FIX fields are categorized into roughly 20 datatypes. A datatype should be defined in terms of its value space, the range of its possible values, not in terms of its lexical space, its encoding format. In fact, a FIX datatype may be mapped to any number of wire formats. (See datatype mappings section below.)

Some fields are specified as a set of valid values. This is known as code set, and it can be thought of as a specialized datatype. (See Code set section below.)

Each FIX datatype is described by a `<datatype>` element, a child of `<datatypes>`. A datatype may optionally inherit properties from a type specified by the `baseType` attribute. For example, `Qty` datatype, used by fields like OrderQty, has `baseType` of `float`, a more generic FIX datatype.

Generally, FIX datatypes need be defined only once and are copied from the baseline standard.

3.5.2 Datatype mappings

A <datatype> element may contain <mappedDatatype> elements corresponding to any number of type systems. Type systems include XML, SBE, GPB, JSON, and ISO 11404, a generic type taxonomy. An XML schema mapping is obviously needed by FIXML.

The standard attribute of <datatype> tells which type system the mapping is for. Its base attribute tells what the FIX datatype maps to in the particular standard. For example, FIX type Qty maps to XML schema type xs:decimal.

The ISO/IEC 11404 General Purpose Datatypes standard contains a taxonomy of language-independent types and enumerates their characteristics. One of the benefits of following this standard is that it will be easier to map FIX data types to other message standards, such as ISO 20022 (SWIFT). Rather than creating numerous one-off mappings to other type systems, is it likely more efficient to map each to ISO 11404 once, and then compare mappings in an associative model to identify the commonalities. (The XML schema standard claims to derive its datatypes from ISO 11404, but mapping to the generic standard is more precise and comprehensive than filtering it through the XML interpretation.)

The lower and upper bounds of a bounded datatype may be set with minInclusive and maxExclusive attributes.

A FIX datatype with mappings to XML schema and General Purpose Datatypes

```
<fixr:datatype name="SeqNum" baseType="int">
  <fixr:mappedDatatype standard="XML" base="xs:positiveInteger"/>
  <fixr:mappedDatatype standard="ISO11404" base="Ordinal"/>
</fixr:datatype>
```

3.6 Code sets

A code set contains the valid values of a data element. Each unique valid value is called a code. In the terminology of ISO 11404, such a data element is called a “state”. It has a finite collection of valid values. (This is distinguished from an enumeration, in which the order of values matters. In a state, each of the values must be unique, but order is not significant. Hence, the values collection is a set.)

In FIX and other protocols, many fields may share a code set. For example, the SecurityIDSource and UnderlyingSecurityIDSource fields share the same valid values, or code set.

A code set has an underlying datatype to tell its range of possible values. Codes may legally be of any type listed in the <datatypes> section, but typically are int, char or String datatypes in FIX. In an XML file, a code value is presented as a string, but it should be actually transmitted in the correct encoding for the datatype of the code set. For example, if the datatype of a code set was “int”, value “27” should be transmitted in the correct wire format for integer 27, not as character “2” and then character “7”.

A <codeSets> element contains any number of <codeSet> child elements. The schema allows multiple instances of <codeSets> containers, each with a unique name attribute. They may be used organize code sets for different realms, such as for different protocols or internal versus external code sets (see below).

3.6.1 Unique names

The names of code sets and datatypes share a common namespace and must be unique within a schema. This constraint is enforced by the XML schema.

3.6.2 Internal code sets

Internal code sets are maintained in a Repository or Orchestra file by the issuer. The <code> elements that belong to the code set are listed explicitly.

3.6.2.1 Codes

An internal <codeSet> is a container for <code> elements. In the schema, each code has a nname attribute to tell its logical name, and a value attribute to tell its value on the wire.

Codes may be added to a code set over time, or existing codes may be deprecated. The history of codes within a code set may be recorded using the pedigree attributes of attribute group entityAttribGrp.

Codes may be documented with an <annotation> element tree.

An internal code set

```
<fixr:codeSet name="SettlInstSourceCodeSet" type="char" >
  <fixr:code value="1" name="BrokerCredit" added="FIX.4.1"/>
  <fixr:code value="2" name="Institution" added="FIX.4.1"/>
  <fixr:code value="3" name="Investor" added="FIX.4.3"/>
</fixr:codeSet>
```

3.6.3 External code sets

In some cases, FIX shares code sets with other protocols. Examples include currency, language, and country codes defined by another standard. This is called an external code set because the valid values are maintained by the external standard, not within the Repository or Orchestra file. To provide a reference to an external standard, use <codeSet> attribute specUrl. Additional references can be supplied with <annotation> elements.

In the case of an external code set, <code> elements are not listed in the Orchestra file.

An external code set. Currency is defined as a FIX datatype with valid values defined by standard ISO 4217.

```
<fixr:codeSet name="CurrencyCode" type="Currency" specUrl="
http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.ht
m?csnumber=64758"/>
```

3.7 Fields

A field carries a specific business meaning (semantics) as described in FIX specifications or other protocol. In the schema, a <field> element is contained by parent element <fields>. There is a single <fields> container, no matter how may versions of FIX protocol are described in a file. By using one list for all protocol versions, duplication is avoided. The collection of fields should be thought of as an append-only list; the id of a deprecated field must not be reused. The pedigree attributes of attribute group entityAttribGrp are used to tell the history of a field, including the protocol version in which it was added.

In FIX, a field has two unique identifiers, numeric id, also known as tag, and a descriptive string name. Both id and name must be unique among all message elements in a schema. Although the two keys may be considered duplication, both attributes must be used in all field references, and it is an error if the values are inconsistent.

Like other message elements, a field may be documented with an <annotation> element tree as described above. Also, the baseCategory attribute may be used to categorize fields. There are several more optional attributes which are described in the message structure section below.

3.7.1 Data domain of a field

Every field must have data domain of either a <datatype> name or more specifically, a collection of valid values specified by a <codeSet> reference. In either case, the domain of a field is specified in its type attribute. The attribute type refers to either a <datatype> element or a <codeSet> element by its name attribute. In the case of a <codeSet>, there is a level of indirection to its type attribute to arrive at a <datatype>.

A field with a code set and another with a datatype domain

```
   <fixr:field id="59" name="TimeInForce" type="TimeInForceCodeSet"/>
   <fixr:field id="60" name="TransactTime" type="UTCTimestamp"/>
```

3.7.2 Data fields

A field of data datatype is variable length. In FIX tag=value encoding, the length of such a field is prefixed by a separate field of type Length. In other encodings, the length is implicit in the presentation protocol. For data fields, the associated Length field is referenced by lengthId and lengthName attributes, references to its id and name attributes, respectively. Both keys must be supplied if field datatype is data.

A data field and its corresponding Length field

```
   <field added="FIX.2.7" id="95" name="RawDataLength" type="Length"/>
   <field added="FIX.2.7" id="96" name="RawData" type="data"
lengthId="95" lengthName="RawDataLength"/>
```

3.7.3 Discriminator fields

FIX contains fields for which its value domain is modified by another field. This is variously called a choice, discriminated union, tagged union, or variant. The field that modifies the range of values of another field is called a discriminator. For example, SecurityIDSource is the discriminator for the values of SecurityID. If the value of SecurityIDSource is 4, then the value of SecurityID must be a valid ISIN, and so forth. This relationship may be indicated in Orchestra by adding the attributes discriminatorId and discriminatorName to a field.

A field modified by a discriminator

```
   <fixr:field added="FIX.2.7"
                  id="48"
                  name="SecurityID"
                  type="String"
                  abbrName="ID"
                  discriminatorId="22"
                  discriminatorName="SecurityIDSource">
         <fixr:annotation>
            <fixr:documentation purpose="SYNOPSIS">
         Security identifier value of SecurityIDSource (22) type (e.g.
CUSIP, SEDOL, ISIN, etc). Requires SecurityIDSource.
      </fixr:documentation>
         </fixr:annotation>
   </fixr:field>
```


3.7.4 Overridable and fixed field attributes

Some attributes of a field, such as minimum and maximum values and length, may be overridden for a particular usage in the message structure that contains a field reference. However, the key identifiers `id` and `name` as well as `type` attribute may not be overridden. It is possible to override which codes of a code set are supported in a particular scenario, however. See the message structure section below.

3.7.5 Field value uniqueness

Some fields are required to be populated with unique values, either globally or within a defined scope. A scope of uniqueness may be specified with reference to other fields.

Uniqueness may be specified with the `<unique>` element within a rule applied to either a field definition or a reference. The uniqueness of a field may be specified for all uses of the field if a rule is placed as a child of a `<field>` element, or it may apply to one use case of the field by placing it within a `<fieldRef>` element.

Globally unique value is required for all instances of a field

```
<fixr:field id="11" name="ClOrdID" presence="required"">
  <fixr:rule>
    <fixr:unique/>
  </fixr:rule>
</fixr:field>
```

A key field provides the scope of uniqueness. Example: unique values per day

```
<fixr:fieldRef id="11" name="ClOrdID" presence="required">
  <fixr:rule>
    <fixr:unique>
      <fixr:fieldRef id="75" name="TradeDate"/>
    </fixr:unique>
  </fixr:rule>
<fixr:fieldRef>
```

A combination of fields defines scope of uniqueness. Example: unique per day and market segment.

```
<fixr:fieldRef id="11" name="ClOrdID" presence="required">
  <fixr:rule>
    <fixr:unique>
      <fixr:fieldRef id="75" name="TradeDate"/>
      <fixr:fieldRef id="1300" name="MarketSegmentID"/>
    </fixr:unique>
  </fixr:rule>
<fixr:fieldRef>
```

3.8 Message structures

3.8.1 Components

A component is a sequence of fields and nested components. There are two types of components, common blocks and repeating groups. Both types are contained by the <`components`> parent element.

Like the messages that contain them, components may be overloaded for slightly different layouts for different scenarios.

3.8.1.1 Component identifiers

Like a field, a component has a numeric `id` attribute and a string `name` attribute. For all references to a component, the two key attributes must be consistent. The schema enforces uniqueness of the `id` attribute among both types of components. Like a field, a component can be annotated for documentation and carries pedigree attributes of attribute group `entityAttribGrp`.

The `scenario` attribute of a component identifiers a use case; multiple components may have the same `name`, but the combination of `name` and `scenario` must be unique. Scenario has a default value of "base", so if a component only has one variation, there is no need to qualify it.

Inheritance is specified with the attribute `extends`. It gives the name of another component scenario to inherit from, possibly "base". When extending another scenario, the component elements of the inherited scenario are assumed to be included.

3.8.1.2 Common block

A common block component is designed to specified once in detail but reused in multiple message types by reference. An example of a common block is "Instrument". It is a collection of the possible fields describing an instrument to be traded, and it is used in many FIX messages. A common block is implemented as a <`component`> element in the schema.

Rules about order of fields or nested components, if any, depend upon the presentation protocol. Since Orchestra supports multiple encodings, the order of fields in an Orchestra file is not guaranteed to match the order on the wire.

3.8.1.3 Repeating group

A repeating group is like a common block but with one additional feature: it represents an *array of* blocks to be sent on the wire. In FIX tag=value encoding, a counter of datatype `NumInGroup` precedes the array when transmitted. In other encodings, such as FIXML, the array is implicit in the presentation protocol.

A repeating group is specified by a <`group`> element. It has attributes to specify the associated NumInGroup field by id and name, `numInGroupId` and `numInGroupName`, respectively. As with any field reference, both keys must be populated and must be consistent with the referenced field.

Limits on the size of a repeating group may optionally be specified with `implMinOccurs` and `implMaxOccurs` attributes. If those attributes are not present, then the repeating has unbound size.

A repeating group with member fields and reference to NumInGroup

```
   <fixr:group id="2096" added="FIX.4.4"
name="UndlyInstrumentPtysSubGrp" numInGroupId="1062"
numInGroupName="NoUndlyInstrumentPartySubIDs" category="Common">
     <fieldRef id="1063" name="UnderlyingInstrumentPartySubID"
added="FIX.4.4"/>
     <fieldRef id="1064" name="UnderlyingInstrumentPartySubIDType"
added="FIX.4.4"/>
   </fixr:group>
```


3.8.1.4 Component members

A component may contain reference elements of three types in any combination. A component must contain at least one member.

- A <fieldRef> element represents a field in a block or repeating group. It is a reference to a <field> element within the <fields> container. The id and name attributes of the reference must match a <field> consistently.
- A <componentRef> element represents a nested component. There is no limit in the schema to the level of nesting, although a presentation protocol may have rules about it, and there may be practical limits. The reference must match the referenced <component> on both id and name attributes.
- A <groupRef> element similarly refers to a nested <group> repeating group element. Limits of the size of particular instance of a repeating group may be overridden by setting implMinOccurs and implMaxOccurs attributes on the <groupRef> element.

A component with two fields and a nested repeating group

```
   <fixr:component name="InstrumentExtension" id="1004"
category="Common" added="FIX.4.4">
     <fixr:fieldRef id="668" name="DeliveryForm" added="FIX.4.4"/>
     <fixr:fieldRef id="869" name="PctAtRisk" added="FIX.4.4"/>
     <fixr:groupRef id="2074" name="AttrbGrp" added="FIX.4.4"/>
   </fixr:component>
```

3.8.1.5 In-line component definition

Normally, fields, components and groups are listed by reference in order to avoid duplication of their attributes. In the case of a field, it may be used in many messages, so a file generally only defines a <field> once in all its glory, while each usage refers to it with a simpler <fieldRef>. Likewise, for components and groups.

However, the schema allows for in-line definition of a <field> contained by a parent <component>. This may be simpler when the usage is known to be a singleton, and therefore, there is no duplication of attributes.

3.8.2 Presence

Each of the members of a component or message, namely <fieldRef>, <componentRef> or <groupRef>, have a presence attribute. The possible values of presence are:

- **required**—the field or component must always be present in a message.
- **optional**—the field or component is optional.
- **conditional**—the field or component is conditionally required. See below.
- **forbidden**—the element is forbidden in a particular scenario described (but may be allowed in others). Scenarios are described below.
- **ignored**—the element is permitted but is not processed by the receiving party, and thus, no validation is performed on it.
- **constant**—the field has a constant value.

3.8.3 Constant field value

A field may be set to a constant value. A specific value of a field is often useful to distinguish scenarios, or use cases for a message type. For example, values of ExecType distinguish various scenarios of

ExecutionReport. Also, if a presentation protocol supports constants, a constant field need not be transmitted on the wire.

A constant field. SecurityIDSource is always code "1" (CUSIP).

```
   <fixr:fieldRef id="22" name="SecurityIDSource" presence="constant"
value="1"/>
```

3.8.4 Default value of an optional field

For an optional field, a default value may be specified if the sender does not provide the field.

An optional field with default value. TimeInForce default is '0' (Day).

```
   <fixr:fieldRef id="59" name="TimeInForce" presence="optional"
value="0"/>
```

3.8.5 Conditionally required field

The presence of a conditionally required field depends upon other fields in a component or message. For example, StopPx is required when OrdType is Stop or StopLimit. If OrdType has any other value like Limit or Market, then StopPx is not required.

The condition that tells when a conditionally required field is required is contained by a <rule> element tree under a <fieldRef>. A <rule> element may contain an override of presence as well as certain other field attributes. Each rule is specified by a <when> element that gives the condition for the override. The XML content (text node) of the <when> element is a conditional expression that follows a grammar described in the conditional expressions section below. The attribute override such as presence="required" attribute is applied to the <when> element.

Rules for a conditionally required field

```
   <fixr:fieldRef id="99" name="StopPx" presence="conditional">
     <fixr:rule name="StopOrderRequiresStopPx" presence="required">
       <fixr:when>OrdType == ^Stop</fixr:when>
     </fixr:rule>
     <fixr:rule name="LimitOrderForbidsStopPx" presence="forbidden">
       <fixr:when>OrdType != ^Stop</fixr:when>
     </fixr:rule>
   </fixr:fieldRef>
```

3.8.6 Message

A message in an Orchestra file describes a unit to be sent on the wire between counterparties.

Like a <component>, a <message> element has id and name attributes. It also has msgType attribute, a short name. In tag=value encoding, msgType is the value of tag 35.

In FIX, a MsgType is often reused for multiple use cases. For example, an ExecutionReport with msgType="8", is overloaded for acceptance of an order, rejection, execution, cancel confirmation, etc. In the Orchestra schema, the scenario attribute is used to name each of those use cases. Each of the variations of a MsgType can have slightly different message structures.

Another attribute of <message> called flow ties a message to an exchange of messages between actors.

3.8.6.1 Message structure

The <messages> element contains any number of child <message> elements. From the perspective of the XML schema, a <message> is very similar to a <component>; they contain the same member types and share most attributes. However, <message> is a top-level entity only; it cannot be contained by other message parts, nor can messages be nested.

Unlike <component>, the parts of a message are contained by a child <structure> element, which in turn holds <fieldRef>, <componentRef> and <groupRef> elements.

A message structure with a field, nested components, and a nested repeating group

```
   <fixr:message name="TradingSessionList" id="100" msgType="BJ"
category="MarketStructureReferenceData" section="PreTrade">
   <fixr:structure>
     <fixr:componentRef id="1024" name="StandardHeader"
presence="required"/>
     <fixr:componentRef id="1057" name="ApplicationSequenceControl"/>
     <fixr:fieldRef id="335" name="TradSesReqID" />
     <fixr:groupRef id="2099" name="TrdSessLstGrp"
presence="required"/>
     <fixr:componentRef id="1025" name="StandardTrailer"
presence="required"/>
   </fixr:structure>
   </fixr:message>
```

3.8.6.2 Message structure extension

Message structures commonly vary with scenario or use case. For example, an ExecutionReport might look quite different in its execution use case versus a cancel-confirmation use case. To avoid duplication of message structures, the XML schema supports declaration of a base structure that is common to all use cases with any number of extensions. The extended structures inherit the base message structure. The attribute that gives names a use case is scenario. If no scenario is explicitly given, it defaults to "base".

Inheritance is specified with the attribute extends. It gives the name of another scenaro to inherit from, possibly "base". When extending another scenario, the message structure of the inherited scenario is assumed to be included. Only added fields need be listed in the <structure> of the extended message scenario. Logically, the extended structure is a conjoined collection; it does not imply anything about the order of fields in the derived message.

3.8.6.3 Responses

Aside from <structure>, <message> has another child element called <responses>; it is explained in the workflow section below.

3.9 Expressions

3.9.1 Conditional expressions

Conditional expressions are rules that are expressed in Domain Specific Language (DSL). They are evaluated by substituting actual values from a message and other state information for tokens in the expression. A conditional expression is of Boolean type. That is, it evaluates true or false. If true, it determines the value of another attribute or that some action should take place, such as sending a certain response message.

Conditional expressions are used in Orchestra:

- To tell when a conditionally required field is required (`presence=required`)
- To tell when a field attribute aside from `presence` is overridden, such as setting the range of valid values with `minInclusive` and `maxInclusive` attributes. It can even tell when to override the `type` of a field. For example, the `type` of SecurityID could be overridden, depending on the value of SecurityIDSource. Some kinds of security IDs are strings while others are numeric.
- To tell when a specific workflow response should be sent or other action taken

All conditions are declared in the XML content of a <when> element. See the Score DSL section below for details of the grammar.

3.9.2 Assignment expressions

Assignment expressions are used to set the value of a field in an outgoing message or to alter a state variable that belongs to an actor. The grammar of assignment expressions is also governed by the Score DSL.

3.9.2.1 Assigning a field

To assign a field in an outgoing message, an <assign> element is placed within the context of a <fieldRef> in the message structure. The content of the <assign> element (text node) contains a Score expression giving the value to set. The value must evaluate to a datatype compatible with the type of the field.

Field assignment: echo the value of a field from an incoming message

```
<fixr:fieldRef id="11" name="ClOrdID" added="FIX.2.7"
updated="FIX.5.0SP2" updatedEP="188">
   <fixr:assign>in.ClOrdID</fixr:assign>
</fixr:fieldRef>
```

3.9.2.2 Assigning repeating group entries

Within the context of a `<groupRef>`, one or more `<blockAssignment>` elements may be used to specify the assignment of fields in entries of a repeating group. Each instance of `<blockAssignment>` will cause another entry to be constructed. Within a `<blockAssignment>`, the syntax for assigning fields is the same as the assignment of an individual field shown above.

Assignment of two entries in a repeating group

```
<fixr:groupRef id="1012" name="Parties" added="FIX.4.3"
updated="FIX.5.0SP2" updatedEP="188">
   <fixr:blockAssignment>
         <fixr:fieldRef id="448" name="PartyID">
               <fixr:assign>"ABC"</fixr:assign>
         </fixr:fieldRef>
         <fixr:fieldRef id="447" name="PartyIDSource">
               <fixr:assign>^GeneralIdentifier</fixr:assign>
         </fixr:fieldRef>
         <fixr:fieldRef id="452" name="PartyRole">
               <fixr:assign>^ExecutingFirm</fixr:assign>
         </fixr:fieldRef>
   </fixr:blockAssignment>
   <fixr:blockAssignment>
         <fixr:fieldRef id="448" name="PartyID">
               <fixr:assign>"DEF"</fixr:assign>
```



```
         </fixr:fieldRef>
         <fixr:fieldRef id="447" name="PartyIDSource">
               <fixr:assign>^GeneralIdentifier</fixr:assign>
         </fixr:fieldRef>
         <fixr:fieldRef id="452" name="PartyRole">
               <fixr:assign>^ClearingFirm</fixr:assign>
         </fixr:fieldRef>
   </fixr:blockAssignment>
</fixr:groupRef>
```

3.9.2.3 Assigning a state variable

To assign the value of a state variable when an event occurs, use the `<assign>` element within a response. The expression contained by the element must refer to a state variable contained by an actor. See the Responses section below.

3.9.3 Field attribute rules

Optionally, a <`rule`> element may be added as a child to <`fieldRef`> to control an attribute of a field dynamically. Multiple rules are allowed for the same field reference to affect multiple attributes or to generate different values of an attribute under different conditions.

The attributes of a <`fieldRef`> that can be controlled by a rule include `type` and any member of `fieldAttribGrp` attribute group. That group includes `presence` and attributes to control the length of a field. A rule about `presence` tells when a conditionally required field is required.

3.9.4 Field validation rules

Orchestra has several ways to specify when a field value is valid. One is to set a field's type to a code set that lists all valid values. Another is to set a valid range using attributes `minInclusive` and `maxInclusive`.

More complex rules can be written under a <`fieldRef`> that reference the values of other fields or the state variables of actors. Rules can be quite dynamic. For example, a market might reject orders with limit price outside a band of some differential above or below the last sale price.

Rule violations can then be captured by a state variable, and if appropriate, an action can be taken, such as sending a reject message. It is important, particularly when generating test systems, to capture all violations rather than reacting to the first one encountered. It is recommended to capture all violations in a repeating group variable of an <`actor`> element. Responses can be defined in the actor to perform actions such as sending a reject message for certain kinds of violations.

A field valuation rule sets a state variable when tripped

```
   <fixr:fieldRef id="44" name="Price">
     <fixr:rule name="tick" >
       <fixr:assign>$validator.ViolationGrp[].ruleViolated=
"tick"</fixr:assign>
         <!-- price not even tick increment of .05 ->
       <fixr:when>(Price * 100) % 5 != 0</fixr:when>
     </fixr:rule>
   </fixr:fieldRef>
```

A state variable to hold rule violations


```
   <fixr:actor name="validator">
     <fixr:group id="10000" name="ViolationGrp" numInGroupId="10001"
numInGroupName="NoViolations">
       <fixr:field id="10002" name="ruleViolated" type="String">
     </fixr:group>
   </fixr:actor>
```

3.9.5 Response conditions

A <when> element with conditional expression is also supported in the <message>/<responses> element tree. See workflow below for usage.

3.10 Workflow

Workflow is the behavior of a FIX party with respect to the exchange of messages. For each received message type, one or more possible actions can be specified under the <message>/<responses> element.

Workflow in Orchestra recognizes that there is not always a 1:1 relationship between a FIX MsgType and a use case. Some FIX message types such as ExecutionReport are overloaded for many different meanings. Therefore, messages in Orchestra are identified primarily by their FIX type, but with a qualification for a specific use case. We call each message use case a scenario.

Behavior may depend upon more information than a receive message itself. External state information enters it as well, e.g. the state of an order book. The <actors> element tree provides a place to store such external state information. An actor can also be used to identify the originator or receiver of a message.

3.10.1 Scenarios

A scenario is one use case of a specific message type, as identified by key attributes name and msgType in messageAttribGrp attribute group supported by <message>. A scenario name is stored in the scenario attribute of <message>. If there is only one use case for a message type, then scenario need not be populated. It defaults to “base”. Scenarios must be unique per message type and it is an error to have more than one <message> element of the same type without a scenario since they would in effect be duplicates of scenario “base”.

This standard imposes no naming convention for scenarios. Implementers are free to choose names that are meaningful in their business.

Each scenario is represented by a <message> element, and thus has its own message contents in its <structure> child element and its own <responses> element tree.

3.10.1.1 Mapping a message to a scenario

This section is non-normative.

The task of mapping an actual received message to a scenario declaration in Orchestra is left to implementations. The first level of matching is on message the msgType attribute. However, that message type may have several scenarios. Pattern matching strategies might include comparing a message to expected required fields, mapping values of a distinguishing field like ExecType to its code set literals, and so forth.

3.10.2 Actors

An <actor> element represents either a counterparty to a FIX session or an external entity that holds state relevant to application and session behavior. An actor can take actions such as assigning state or transitioning a state machine. If it represents a session counterparty, it can send FIX messages. Also, actions can be time dependent. An Orchestra file may declare any number of actors within the <actors> parent element. The name attribute of an <actor> element must be unique within an Orchestra file.

3.10.2.1 State variables

Actors can hold state variables in the form of FIX fields. That is, each state variable has an id and name for identification and a value of a FIX datatype. Like any field, valid values can be constrained to a code set or range. The datatype or code set is declared in the type attribute, just like any field

If a state variable corresponds to a standard FIX field, it can be declared as a <fieldRef> element child of the <actor>. Alternatively, it can be declared in-line as a <field> element. Additionally, state variables can be organized as components or repeating groups.

A state variable can be tested in a conditional expression or set by an assignment expression.

An actor with state variables

```
<fixr:actor name="Market">
  <fixr:fieldRef id="336" name="TradingSessionID"/>
  <fixr:fieldRef id="75" name="TradeDate"/>
</fixr:actor>
```

3.10.2.2 State machines

A state machine has discrete values called states and defined transitions between states. A state machine is declared in XML as a <states> child element of an <actor>. The <states> element contains any number <state> children, and one initial state of the state machine, as <initial> element. It is an error to declare more than one initial state. Some state changes are allowed and others disallowed; changes can only be made through explicitly declared transitions. A <transition> child of a <state> or <initial> gives the name of the new state of the state machine in its target attribute.

States and transitions must have unique names within a state machine.

The current state of a state machine can be tested by a conditional expression, and a transition can be invoked by an assignment expression.

A state machine for market phases


```
<fixr:states name="Phase">
  <fixr:initial name="Closed">
    <fixr:transition name="Reopening" target="Preopen"/>
  </fixr:initial>
  <fixr:state name="Halted">
    <fixr:transition name="Resumed" target="Preopen"/>
  </fixr:state>
  <fixr:state name="Open">
    <fixr:transition name="Closing" target="Preclose"/>
  </fixr:state>
  <fixr:state name="Preopen">
    <fixr:transition name="Opened" target="Open"/>
  </fixr:state>
  <fixr:state name="Preclose">
    <fixr:transition name="Closed" target="Closed"/>
  </fixr:state>
</fixr:states>
```

3.10.2.3 Timers

Some application layer and session layer behavior is time dependent. An event can fire when a timer expires to affect other states or send a message.

Like a state machine, a <timer> is the child of an <actor>, and it has a name attribute.

3.10.3 Flows

A <flow> element represents a stream of messages from one actor (source) to another (destination). A flow depends on abstractions of the session and transport layers, but is an application-layer view of message exchange behavior. It is intended to be session and transport protocol independent. Multiple application flows may be multiplexed in a FIX session.

A <flow> is identified by its name attribute. It must have a source and a destination attribute, and both of those must match the name of an <actor> element. The messageCast attribute defaults to unicast, but may be set to multicast.

The optional reliability attribute describes the delivery guarantee of messages on the flow. It takes one of these values:

- **bestEffort**—no delivery guarantee
- **idempotent**—deliver at-most once
- **recoverable**—deliver exactly once

Example of actors and flows

```
<fixr:actors>
   <fixr:actor name="BuySide"/>
   <fixr:actor name="SellSide"/>
   <fixr:flow name="OrderEntry" source="BuySide" destination="SellSide"
messageCast="unicast" reliability="idempotent"/>
   <fixr:flow name="Executions" source="SellSide" destination="BuySide"
messageCast="unicast" reliability="recoverable"/>
   <fixr:flow name="MarketData" source="SellSide" destination="BuySide"
messageCast="multicast" reliability="bestEffort"/>
</fixr:actors>
```

3.10.4 Responses

Responses to a received message can be of these types:

- A message is sent in reply to the received message
- A state variable is changed
- A state machine transition is invoked
- A timer is started or canceled

Multiple responses can be specified for a given message scenario as children of its <responses> element.

3.10.4.1 Message response

A < messageRef> child of response represents a reply to the received message. Its name, msgType and scenario attributes are the combined key to a matching <message> to send.

Send a response message

```
   <fixr:messageRef name="ExecutionReport" msgType="8" scenario="booked
"/>
```

3.10.4.2 State variable response

An <assign> element changes the value of a state variable belonging to an actor. Its child element <assign> references one or more state variables to change in the form of an assignment expression.

Assign a state variable belonging to actor "participant"

```
   <fixr:assign>$participant.RiskLimitAmount=15000</fixr:assign>
```

3.10.4.3 State machine response

A < trigger> element represents a state machine transition invoked when a message is received. Its statemachine attribute identifies the name of the state machine, and name attribute refers to the name of a <transition> within that state machine.

Invoking a state machine transition: the market resumes after a halt

```
   <fixr:transitionRef actor="market" stateMachine="phase"
name="resume"/>
```

3.10.4.4 Timer operation response

A <timerSchedule> element invokes an operation to either start or cancel a timer. The name attribute refers to the name of the timer, operation tells whether to start or cancel, and interval gives the

elapsed time. Interval is expressed in the lexical space of XML schema type `duration`. That type includes the magnitude and time unit of the period in conformance to standard ISO 8601. The <`responses`> elements represents actions to take when the timer expires. Actions can include sending a message, setting a state variable, or invoking a state machine transition.

Invoking a timer operation: start a timer for 120 seconds. On timer expiration, send a cancel message, provided the order is still open

```
   <fixr:timerSchedule actor="trader" name="exposureTimer"
operation="START" interval="PT120S">
   <fixr:responses>
   <fixr:response>
     <fixr:messageRef name="OrderCancelRequest" msgType="F">
   </fixr:response>
   <fixr:responses>
   </fixr:timerSchedule>
```

3.11 Semantic Concepts

Optionally, semantic concepts may be identified by name, even when the encoding of such a concept changes across versions of a protocol. Anchoring a changing encoding to stable concept can be used to inform applications such as message translators.

A concept may be tied to a field or group fields. Values may be variable or constant. Also, a concept name may be used to link a generic event to its message type and scenario.

Recognized concept names will be published by FIX Trading Community. To maximize portability, users should conform to those names.

The semantic concept is stable, but the FIX 4.2 fields were replaced

FIX 4.2 encoding

```
   <fixr:concept name="ProgramOrderMember">
         <fixr:fieldRef id="47" name="Rule80A" presence="constant"
value="D"/>
   </fixr:concept>
```

FIX 4.4 encoding

```
   <fixr:concept name="ProgramOrderMember">
         <fixr:fieldRef id="528" name="OrderCapacity"
presence="constant" value="P"/>
         <fixr:fieldRef id="529" name="OrderRestrictions"
presence="constant" value="12"/>
   </fixr:concept>
```


The name of a message changed
FIX 4.2 encoding

```
   <fixr:concept name="BaseOrder">
         <fixr:messageRef name="OrderSingle" msgType="D"
scenario="base"/>
   </fixr:concept>
```

FIX 4.4 encoding

```
   <fixr:concept name="BaseOrder">
         <fixr:messageRef name="NewOrderSingle" msgType="D"
scenario="base"/>
   </fixr:concept>
```

3.12

4 Interfaces XML Schema

4.1 XML Schema (XSD)

The FIXInterfaces schema represents service offering and session provisioning. Its XML namespace is `"http://fixprotocol.io/2016/fixinterfaces"`.

4.1.1 Conformance

All published Interface files **must** conform to the standard XML schema. This can be validated with common XML parsers and related tools.

4.1.2 Schema location

The XML schema is currently available in GitHub project fix-orchestra module interfaces2016. Upon promotion to draft standard, it will be made available at a URI consistent with its XML namespace.

4.1.3 Root element

The root element an Interfaces XML file is <`interfaces`>. This snippet shows that element with required namespaces:

```
<fixi:interfaces xmlns:dcterms="http://purl.org/dc/terms/"
xmlns:fixi="http://fixprotocol.io/2016/fixinterfaces"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://fixprotocol.io/2016/fixinterfaces
FixInterfaces2016.xsd">
```

4.1.4 Supplementary documentation

See the separate document “FixInterfaces2016.html” for a detailed technical reference for the Interfaces XML schema. The remainder of this section serves as an overview and explains motivations for the design.

4.1.5 Protocol relationship

The schema was primarily designed to describe metadata about FIX protocols. However, it was also intended to be generic enough to work with other common financial industry protocols, especially when FIX is used in combination with other protocols.

4.1.6 Extensibility

This schema was designed to maximize extensibility so it that represent a wide range of applications, even with non-FIX protocols. Most elements allow addition of attributes, and types allow additional child elements, possibly conformant to other XML schemas.

4.1.7 Provenance

The `<metadata>` element is used to identify a particular Interfaces file and the issuer of that file. It can contain any of the elements defined by the Dublin Core XML schema. Recommended elements include publisher, date, and rights.

4.1.8 Session effective times

Optionally, a session may be configured for start and end time. Adding a session prior to its effective time allows configuration tasks to be carried out in advance of usage.

4.1.9 Incremental changes

The Interface file format represents current state. Aside from session effective times, it does not carry full pedigree. However, if a party wishes to represent incremental changes to a file, it may do so using XML patch operations as specified in IETF RFC 5261.

4.1.10 Interface

The root element `<interfaces>` contains one or more `<interface>` elements. An interface is a collection of protocols and services exposed by a counterparty. An interface may be configured for one or more service offerings and all the protocols that make up a communication stack. A service offering is exposed as a `<service>` element, and protocols are given as elements for each layer of a stack. Also, an interface may contain any number of session configurations under its child `<sessions>` element. An <interface> element has a `name` attribute.

4.1.11 Protocols

An <interface> element has children for each layer of a protocol stack. The children are `<userInterface, <encoding>, <sessionProtocol>, <transport>`, and `<protocol>`. An `<interface>` may have multiple instances of a protocol. For example, a session may use primary and secondary transports.

Any message-oriented protocol may have an `orchestration` attribute that consists of a URI. It is a link to an Orchestra file that describes message structures and workflow. A URI may link to a web resource or a local file.

All the protocol elements have `name` and `version` attributes.

4.1.12 Service

A service is an application layer protocol. The `<service>` element is of XML `protocolType`, carrying the same attributes as other protocols.

4.1.13 Transport

The <transport> element is derived from XML protocolType but has additional attributes address, messageCast and use. The optional messageCast attribute has an enumeration of values: unicast, multicast and broadcast. The optional use attribute can have values primary, secondary and alternate.

4.1.14 Session

A <session> inherits services and protocols from its parent <interface>, but it may have further refinement or overrides of protocol settings, such as a transport address.

A session has one or more identifiers in child <identifier> elements. The <value> child of <identifier> may be of any XML type, even an element tree.

5 Score DSL

5.1 Grammar

5.1.1 Comments

Comments may be inserted in Score expressions in two forms. Comments are ignored by an expression evaluator but give a humanly readable explanation.

5.1.1.1 C-language style comments

C-language style comments are contained by tokens `/*` and `*/`.

```
/* This is a C style comment. */
```

5.1.1.2 Line comments

Line comments extend from the token `//` to the next line break.

```
// This is a line comment.
```

5.1.2 Literals

A literal stands for a value that is assignable to a FIX datatype.

5.1.2.1 Character literal

A character literal is of FIX datatype `char`. It is delimited by single quotes.

Example: `'a'`

5.1.2.2 String literal

A character literal is of FIX datatype `String`. It is delimited by double quotes.

Example: `"A String literal"`

5.1.2.3 Integer literal

An integer literal is of FIX datatype `int`. It is a sequence of digits, such as `123`.

An integer literal may be preceded by a hyphen character that represents the unary minus operator, such as `-123`.

5.1.2.4 Decimal literal

A decimal literal is assignable to FIX datatypes `float`, `Price`, `Amt`, `Qty`, `PriceOffset` or `Percentage`. It is a sequence of digits followed by a decimal point (period character) and another sequence of digits. At least one digit must precede and follow the decimal point. A decimal literal may be preceded by a unary minus operator (hyphen character).

Example: `123.456`

5.1.2.5 Date-time literals

Date, time of day, and date-time literals are delimited by the `#` character. The syntax within the delimiters is governed by standard ISO 8601 “Date and time format”.

5.1.2.5.1 Date literal

A date literal is of the form YYYY-MM-DD with a hyphen character separating the year, month and day parts. A date literal is of FIX datatype `UTCDateOnly`.

Example date: `#2017-03-21#`

5.1.2.5.2 Time literal

A time of day literal is of the form THH:MM:SS.FFFFFFFFFTZD with a colon character separating the hour, minute and optional second parts. An optional fraction of a second follows a decimal point (period character). It may represent nanosecond precision. Finally, a time literal contains a timezone designator, either the literal Z, or a timezone offset from UTC. A timezone offset is of the form [+|-]HH:MM. It represents an offset from UTC in hours and minutes. A time literal is of FIX datatype `UTCTimeOnly`.

Example times:

```
#T09:58:24.123456789Z#
#T09:58:24Z#
#T09:58-06:00#
```

5.1.2.5.3 Date-time literal

A time of day literal is of the form YYYY-MM-DD THH:MM:SS.FFFFFFFFFTZD. The syntax of the parts are the same as a date literal followed by a time literal. A date-time literal is of FIX datatype `UTCTimestamp`.

Example times:

```
#2017-03-21T09:58:24.123456789Z#
#2017-03-21T09:58:24Z#
#2017-03-21T09:58-06:00#
```

5.1.2.5.4 Duration literal

A duration literal is of the form PYMWDTHMS. In all cases, ‘P’ is a prefix, and ‘T’ separates date units from time of day units. The units of time are represented by literal Y=year, M=month, W=week, D=day, H=hour, M=minute, S=second. Each unit is optional, but they may be used in any combination.

Currently, there is no FIX datatype that represents duration, but a duration literal may be used with date and time literals in date and time expressions in the DSL.

Example durations:

7 days: `#P7D#`

1 hour 30 minutes: `#PT1H30M#`

10 seconds: `#PT10S#`

N.B. Month duration is recognized by the ISO 8601 syntax, but since months are of different numbers of days, the resulting duration is indeterminate without some context about how to count days.

5.1.2.6 Code literal

A code of a code set, is designated by its `Namename` preceded by the `^` character. The code set that contains the code is generally inferred by a field scope within an expression.

Example code of OrdType code set: `^StopLimit`

5.1.3 Variables

A variable is named value that is independent of sent and received messages. A variable has a name and a value of any FIX datatype. A state variable is created and populated by an assignment expression (see below). The datatype of a state variable is set by the assignment.

5.1.3.1 Variable names

The following entities must have distinct names to be used in conditional or assignment expressions.

- A field used as a state variable of an actor.
- The current state of a state machine, belonging to an actor.
- A timer that belongs to an actor.

Variable names are always prefixed by the character `$`. Any meaningful name may be used; there is no need to conform to FIX message element names. All names must begin with a letter, and the rest of the name may contain upper or lower case letters, digits, or the underscore character. A name may consist of multiple qualifiers, each separated by a dot (period character). The first qualifier should correspond to an actor name. Variables may be grouped within actor context by further qualifiers.

Example of a variable name: `$myactor.totalQty`

5.1.4 Message element references

The DSL syntax allows access to fields in received messages and population of field in messages to be sent.

5.1.4.1 Field names

The following entities must have distinct names to be used in conditional or assignment expressions.

- A field at the root level of a message
- A field contained by a repeating group. In the case of a repeating group, an entry is indexed or an entry may be selected by a conditional expression (see below).

The high-level qualifier for a received message is `in`, and the high-level qualifier for an outbound message is `out`. In implementations, the scope of a message may be implicit, making the high-level qualifier unnecessary. Then a field can simply be referenced by name. However, qualification is needed if for example, an expression about a field in an outgoing response message refers to a field in its inbound request.

Example: `TradSesStatus`

5.1.4.2 Repeating group entry selection

If a field is within a repeating group, then an entry in the group must be selected to retrieve the value of the field. This can be done in two ways. The first way is by using a one-based index (ordinal number) to select an entry. The index is surrounded by square brackets. The repeating group and field names are separated by a dot (period character).

Example refers to the MDEntryType field in the second entry of its repeating group:
`MDIncGrp[2].MDEntryType`

The second method of accessing a repeating group entry is by using an equality expression using a second field in the group as a key. The expression is placed in square brackets. The condition selects a repeating group entry by testing equality of a named field in the group entry to a literal value.

Example references PartyID field in the repeating group entry for which PartyRole equals 4:
`Parties[PartyRole==4].PartyID`

Alternatively, the same field can be accessed by using a code literal in the equality expression. In this example, `ClearingFirm` is the name of the code for which the value is integer 4.

```
Parties[PartyRole==^ClearingFirm].PartyID
```

5.1.5 Conditional expressions

Conditional expressions are used for multiple purposes in Orchestra:

- To tell when a conditionally required is in fact required
- To tell when a certain response to a received message is triggered
- To select an entry in a repeating group using a field value, as described above

Conditional expressions take several forms:

- Relational expression: Compare a field's value in a received message to a literal of the field's datatype or a code designated by its `Namename` in a code set associated to the field.
- Relational expression: Compare a field's value in a received message to the value of another field in the same message or to a field used as a state variable belonging to an actor.
- Relational expression: compare the current state of a state machine to a literal representing one of its possible states.
- Compound relational conditions joined by "and" and "or".
- Relational expressions may express set inclusion or data range inclusion.
- Relational expressions may depend on simple expressions that use arithmetic operators on terms.

5.1.5.1 Relational operators

These are the relational operators of the Scope grammar. Operands must be of the same or compatible datatypes.

Token	Name
< or `lt`	less than
<= or `le`	less than or equal
> or `gt`	greater than
>= or `ge`	greater than or equal

5.1.5.2 Equality operators

These are the equality operators of the Scope grammar. Operands must be of the same or compatible datatypes.

Token	Name
== or `eq`	equals
!= or `ne`	not equals

5.1.5.3 Logical operators

These are the logical operators of the Scope grammar. Operands must be Boolean.

Token	Name
`&&` or `and`	and
`||` or `or`	or

5.1.5.4 Arithmetic operators

These are the arithmetic operators of the Scope grammar. Operands must be numeric. Multiplication and division have higher priority than addition and subtraction when not grouped by parentheses.

Token	Name
`*`	multiplication
`/`	division
`%` or `mod`	modulo
`+`	addition
`-`	subtraction

5.1.5.5 Unary operators

These are the unary operators of the Scope grammar.

Token	Name	Operand type
`-`	minus	numeric
`!`	logical not	Boolean

5.1.5.6 Parentheses

Terms of an expression may be grouped by parentheses to override the default operator precedence. Opening and closing parentheses must always match.

5.1.5.7 Set operator

The set operator tests whether a value is in a set of values of the same datatype. The result of the operation is Boolean. The syntax is as follows:

value `in {`member, member …`}`

Value may be a literal, state variable or reference to a message field. Each member may be a literal or other expression of the same datatype.

5.1.5.8 Range operator

The range operator tests whether a value is in a range of values of the same datatype. The result of the operation is Boolean. The syntax is as follows:

value `between` min `and` max

The range operator is a shortcut for value <= min and value >= max. Value may be a literal, state variable or reference to a message field. Min and max may be a literal or other expression of the same datatype.

5.1.5.9 Existence operator

The existence operator tests whether a variable has been defined, or if an element is present in a message. The test is of Boolean type and may be combined with other logical operators to form a conditional expression. The syntax is of the form:

`exists` variable

5.1.6 Assignment expressions

The following entities may be assigned values in an expression:

- A field of a message to be sent in a scenario.
- A field used as a state variable of an actor.

The assignment operator is the = character. The left-hand operand must be either a state variable or a mutable message field. Literals are immutable so a literal cannot be the lvalue of an assignment expression. The right-hand operand can be any expression of a compatible datatype. In the case of a variable, it takes the datatype of the expression if it has never been assigned before.

5.2 Syntax errors

Implementations should throw an exception if a DSL expression does not follow the syntax described above. For example, parentheses are mismatched.

5.3 Semantic errors

Implementations should throw an exception in these cases:

- Operands are of incompatible datatypes. For example, a relational operator is asked to compare a Price value to a UTCTimestamp.
- A variable or message element referenced by an expression does not exist.

6 Semantic Representation

[OWL representation to come]

7 Usage Guidelines

[to come]

8 Examples

Example Orchestra files are provided in the GitHub project FIXTradingCommunity/fix-orchestra.

9 Appendix

9.1 Changes from Repository 2010 Edition

- New features: metadata with provenance, actors, workflow, semantic concepts.
- An Orchestra file supports a single protocol version. If an organization supports multiple protocols, it should issue an Orchestra file for each.
- The codeSets element is now top-level while in the 2010 Edition, enumerations of valid values were contained by fields. This change was made to recognize that code sets may be shared by many fields and also, they may be managed by an external standard.
- The datatype element was enhanced to support mapping FIX datatypes to any other type system, and not just XML schema datatypes.
- The Orchestra schema provides a feature to explicitly link a field, called a discriminator, that modifies the value space of another field.
- Scenarios overload the layout of a message or component for different use cases.

9.2 Compliance

To be useful, various implementations of FIX Orchestra must interoperate. The FIX Trading Community discourages implementations that deviate from this specification while promoting those that are compliant.

At minimum, a compliant application:

- Must conform to the XML schema published in the GitHub fix-orchestra project.
- Must conform to the DSL grammar published in the GitHub project.

Additional compliance utilities may be published. Only applications that pass these checks will qualify for endorsement.