Beehive Object Model

Oracle Corporation

Editors: Terry M. Olkin, Eric S. Chan, Rafiul Ahad

Version 1.0: 01/12/2009 - Created for OASIS ICOM TC

Version 1.1: 03/24/2009 - Conference to subclass Folder

This document describes the object model for Beehive. The document does not attempt to define the model in terms of an actual programming language. Instead, we describe the model using a concise abstract notation. Once we have specified the abstract model, we will be able to define a programming language binding to the model.

1.0 Vocabulary

This section forms a glossary for all the relevant words that we will use throughout the document.

Access Control	A specific type of policy that determines whether a principal may perform a specific operation on an entity or an action on the system.
Action	1. An action is something performed by an actor that may or may not be directed at any particular entity. These actions are subject to access control. An operation is an action on an entity. 2. An Action can be the THEN part of the IF/THEN rule construct. Such a rule action is executed when the rule condition is satisfied.
Actor	An entity that can perform actions upon other entities.
Address Book	A container of contacts.
Agent	An external person or external resource with whom the system can communicate by messages.
Alert	An urgent message to one or more entities that typically requires the immediate attention of the receivers.
Announcement	A message for a group that is posted and that is valid for a specified period of time.
Artifact	An entity that is the result of a communication or collaborative activity. Document creation is an example of collaborative activity resulting in an artifact (the document).
Attachment	An artifact that is associated with some other entity. It represents the association between an attachment holder and an attachable.
Authentication	The process of proving one's identity.
Authorization	The process of checking if an actor has the necessary privileges needed to perform an operation on an entity or an action in the system.
Calendar	A container of possibly recurring schedules; a schedule has a start time and duration such as a meeting, an appointment, or a booking of a resource.
CEN	The collaboration entity name which uniquely identifies an entity in the system.
Chat	Specifically a many-on-many (group), synchronous, usually text-based conversation. *Contextual chat* includes the ability to pass interactive, non-textual information between participants (i.e., chat within the context of one or more artifacts).
Coexistence	A registry that defines how an actor's mail, calendar, task, and address book may coexist in another system for the purpose of seamless access or synchronization to/from Beehive system.

Collaborate	Two or more actors working together for a common objective, usually sharing and producing artifacts.
Community	A group of actors and agents that share common workspaces and policies.
Conference	An event that involves one or more concurrent conversations.
Conference Participant	An actor that is a participant in at least one conversation of a conference.
Contact	(n) An artifact that refers to an addressable entity or a set of entity addresses. It is an entry in a container called *Address Book*. (v) To communicate with the addressable entity.
Contact List	A set of contacts. The set can optionally have internal structure (e.g., hierarchy). Contact lists can be used anywhere a grouping of contacts is needed. See *Address Book*.
Container	A generic set of entities. It is used when the specific type of container is unimportant. We use Folder to represent a specific type of container. See *Folder.*
Context	A frame of reference containing the relevant entities and relationships in which some operation is to be executed. Note that a context need not map directly to an entity - it is often a dynamic grouping. See *Environment*.
Conversation	A multi-directional flow of messages and/or streams among entities. The content of the messages and streams are typically semantically related although this is not a requirement. Conversations have well-defined starting points in time and typically also have ending points (but are not required). A single conversation can be carried out using one of or a multitude of media types.
Conversation Originator	An entity that begins a conversation, typically by sending a message or stream to at least one receiver.
Conversation Participant	An entity which is at times optionally a sender or a receiver of at least one message or stream in a conversation.
Delegate	(n) An actor that can act on behalf of another actor; An actor who has taken on the identity of another actor. (v) The act of granting the ability to act on an actor’s behalf or take on another actor’s identity.
Delete	To remove or mark for removal an artifact from its container. An object that is deleted is technically still present, but is no longer accessible except through extraordiary means.
Device	A terminal from which a person can interact with the system, either to collaborate, communicate or manage resources. There is no assumption as to how the device is connected to the system.

Directory	A container of mappings from entity identifiers (usually referred to as *names*) to the corresponding entity. The mappings form an organization of the entities and their hierarchical relationships. A typical implementation does not provide this mapping directly. Instead, the identifier maps to information that describes the entity that can be used to locate or communicate with it. There may be multiple entries that correspond to a single entity where each entry provides a different name (an alias) for the underlying entity.
Discussion Thread	A conversation.
Distribution List	A contact list whose contacts are interested in receiving information on the same topic.
Document	A specific type of self-contained artifact (a file usually) that represents the result of certain end user applications (e.g., a word processor).
Domain	A name assigned by an Internet registry authority to represent the name of an enterprise or institution. Typically refers to an email scoping. Should only appear in context, e.g., email domain.
EID	An entity identifier which is the immutable part of the collaboration entity name (CEN).
Email	A system for sending and receiving messages electronically over a computer network or a message sent or received by such a system.
Email Thread	A conversation using email as the communications protocol.
Enterprise	The top-level container of all entities. Normally, a single deployment will host only a single enterprise. However, in some cases (e.g., ASP model), a single deployment can host multiple enterprises - but these enterprises are still logically isolated from each other. Enterprises hold organizations.
Entity	A tuple (object) that can be uniquely identified by its collaboration entity name (CEN) which includes a globally unique identifier (**guid**) representing the entity ID (EID). The entity has an optional human readable name. Some parts of the CEN are mutable, but the EID is immutable - EID is created when the entity is created and can never be changed, duplicated or re-used. An entity's name is mutable. Virtually all manipulable objects are entities.
Environment	A frame of reference that may contain related and unrelated information. Typically, an operation is performed on an entity in a context in an environment. The operation may need information from the environment, e.g., time, location, network, in order to perform the operation. Generally speaking, there is only one environment although there are many contexts.
Erase	The act of permanently removing a deleted artifact. Once erased, the artifact is irretrievable.

Event	An event is an occurrence of some action in the system or operation on an entity associated with a well-defined trigger condition.
Event Type	The metadata that defines the name and properties of a class of events.
Fax	(v) To send an image via phone lines using a Fax protocol. (n) An image that was transmitted via phone lines using a Fax protocol. A fax is typically delivered on paper, but need not be.
File	An artifact that contains data and corresponding meta-data that describe the contents (e.g., type, name, etc.).
Folder	A special artifact that is a container for other entities. Types of folders include calendars, address books and end-user folders. Folders may themselves be elements creating hierarchies.
Form	An entity that collects input from actors in a structured way.
Group	A way to name a set of actors.
Hide	Removing an artifact from the view of an actor. That is, the artifact appears to be deleted from the viewing actor's perspective. But the artifact is still present to other actors. The artifact can be unhidden as well. Note that hiding an artifact is merely a viewer preference and has no other effect on the artifact. Contrast this with deleting an artifact or preventing the appearance via access control rules.
Inbox	An entity that contains artifacts with the intent that these artifacts are awaiting some kind of action.
Instant Message	(IM) Specifically a one-on-one, synchronous, usually text based, conversation. Also used as a verb, e.g., IMing, meaning the act of instant messaging.
Journal	An artifact that records an action or event. It has a start time and a duration. Journal entries can be automatic based on user actions or manual (e.g., a "captain's log").
Label	(n) An name that can be directly attached to an entity for the purpose of classifying the entity. A single label can be applied to any number of entities and any entity can support any number of labels. (v) The act of applying a label.
Location	An attribute of an entity describing its logical or physical position in space.
Media Types	The different ways that data can be encoded.
Meeting	An occurrence where collaboration occurs within some context.
Message	A unit of conversation. A message is an artifact that originates at a sender. A message typically contains information, data and/or meta-data. Every message originates at a location at a specific time.

Most-Active	A way to describe an entity with a high frequncy of access by actors.
Name	A “familiar” moniker for some entity.
Notification	A type of message that alerts an actor as to the occurrence of some event.
Observer	A pattern that defines one-to-many dependency between a source entity (for a subset of its event types) and the policies that specifies the plan of actions.
Occurrence	An action, meeting, conference, or happening. An occurrence may reference Occurrence Participants. Occurrences are typically named and have some temporal aspect, minimally a start time.
Occurrence Participant	An actor that participates in an occurrence.
Operation	An action on an entity.
Organization	A container of workspaces and sub-organizations. An organization is the next level container under an enterprise.
Permission	A privilege granted on an entity.
Person	A user that maps to a human being.
Policy	A plan of actions governing the life cycles of entities. A policy can be represented by rule-based applications or workflows.
Preference	A preference declares the choices, desires, course of actions, or customizations of an actor, a group of actors, or a scope.
Preference Profile	A preference profile is a grouping of preset preferences that can be selected for different circumstances. For example, a user can switch back and forth between the regular preference profile and the business travel preference profile.
Presence	The knowledge of an actor’s location, connectedness, status, and ability to converse at a given point in time.
Presentity	A presentity combines devices, services, willingness, and person information for a complete picture of a user’s presence status in the collaboration system.
Principal	A principal contains information about the identities, credentials, and delegations of a particular user for the purpose of access control.
Priority	A ranking of the importance of entities in a set. The higher the priority, the more important the entity. When the entity is a message and the priority is high, we say it is an urgent message.
Privilege	The token for some operation that can be performed by an actor.

Receiver	An entity that receives a message or a stream.
Resource	An allocatable entity limited in capacity for performing an action or being acted upon (i.e., usually not sharable at one time).
Role	A named set of permissions. Actors are then assigned to roles.
Rule	An instruction of the form IF/THEN. A rule is evaluated to either true or false by evaluating some condition. Appropriate action is taken depending on the result.
Scheduling	The act of placing an event on a calendar or allocating a resource using a calendar.
Scope	A scope defines a logical region or neighborhood in the universe of entities that is governed by a common set of policies, which include memberships, roles, and groups. Enterprises, organizations, and workspaces define the boundaries of hierarchies of scopes.
Sender	An entity that originates a message or stream.
Stream	A continuous flow of data in a conversation. A stream originates at a sender and can be distributed to one or more receivers. Every stream starts at a location at a specific start time.
Subscription	A user instruction to have a certain action taken if an event and a given set of conditions occur. It allows a user to prescribe how to automatically react or notify the user when some events occur on an entity. Conditions involve pattern matching expressions on the contents of the events.
Survey	A specific type of form for the purpose of gathering feedbacks on a subject. Related to a *poll* (not defined).
Synchronous Messaging	Communications between a sender and receiver with temporal concurrence. Most real-time communications are examples of synchronous messaging.
System	The totality of actors, entities, and executable code that define one logical instance or installation of the product.
System Actor	A special actor that represents a privileged, non-human user of the system. These are things like processors or services themselves, when they need to act on the system. System actors, because they are usually native to the system or their actions are programmed into the system, are often not authenticated but may be subjected to access control.
System Administrator	A person that can perform privileged operations that effect the system and its environment.
Task	An event that is actionable by an actor at some time.

Template	Meta-data that describes a re-usable structure of a set of entities.
Time	A fourth dimension to location. The temporal nature of an entity. Used both in its absolute form as well as in a relative form.
Time Zone	A geographical region within which the same local time is used.
Transcript	A durable record of a conversation.
Trigger	(n) A stimulus that can set off an event. (v) To generate an event from the trigger point in an entity.
Trigger Point	A distinguished point in the entity representing a source of events. An entity may contain several trigger points that generate events when corresponding operations are performed on the entity.
Urgent	A type of message conveying a sense of pressing importance.
User	An actor that is not a System Actor, usually a human. Users can be in many states including authenticated or not authenticated, on-line or off-line, etc.
Version	The state of an entity at some point in time. Note that not all entities are versionable (even if they change state over time, we don't always refer to the new state as a new version of the entity). Usually a (new) version is established once an entity enters a newly consistent state from its last consistent state.
Voice Mail	A stored message in voice form.
Watch List	A list of entities being monitored. Depending on the type of monitoring, there may be various specializations of watch list.
Wiki Page	A wiki page is an artifact that is continuingly being revised. The revisions are always versioned and maintained.
Workflow	A set of instructions that cause actions to be taken based on events. Once a workflow is instantiated, it usually maintains state. Workflows can execute over extended periods of time (and typically involve user interaction along the way).
Workspace	A long-lived, named entity that defines a context and is a place to collaborate.

2.0 Object Formalisms

2.1 Notation

This section briefly describes the notations used to define the objects and the relations between them.

The statement of the form "*X* => W " is a class definition, which defines the class *X*. *W* is a set of zero or more attribute definitions. The order of attribute definitions within a class is not significant. Each attribute definition is a term of the form "*A* : *Y*," where *A* is an attribute name and *Y* is a type of the attribute which can be a class name or a primitive type. For example, "*Entity* => Name : **string**" specifies that *Entity* has a Name attribute which is of the primitive type **string**. We use square brackets around the attribute name, such as "[*A*] : *Y* " to indicate that it is optional for attribute *A* to hold a value. For example, "*Entity* => [Name] : **string**" specifies that Name is optional for entities[1].

The statement of the form "*C* ::= *X*" defines that "*X* is a *C*" where *C* is a class. It implies that *X* is a subclass of *C* and would inherit all attributes of *C* if defined. For example the statement "*Entity* ::= *Artifact*," where *Entity* is a class, specifies that *Artifact* is a subclass of *Entity*. The formalism includes multiple inheritance. When two or more statements are provided, such as "*B* ::= *D* " and "*C* ::= *D* " which involve the same class *D* with two different super-classes *B* and *C* (*B* and *C* typically do not inherit from each other), then *D* is said to be a subclass of *B* and a subclass of *C*. The subclass *D* will inherit all attributes of *B* and all attributes of *C*. The attributes are uniquely named to avoid collision from multiple inheritance. We also ensure that when a class is derived from multiple super-classes, i.e. involve multiple inheritance, only one of the super-classes is an *Entity*, and the rest of the super-classes are auxiliary classes. An auxiliary class, typeset in *italic Century Gothic font in blue color*, is a class which is not derived from *Entity*. For example, the following set of class derivations specifies that *Scope* is a subclass of *Entity* and the three other super-classes, namely *Bondable*, *Addressable*, and *Container*, which are auxiliary classes.

```
Entity ::= Scope
Bondable ::= Scope
Addressable ::= Scope
Container ::= Scope
```

The auxiliary classes can subclass from other auxiliary classes. For example, in the following class derivations, the super-class *Identifiable* is an auxiliary class.

```
Identifiable ::= Bondable
Identifiable ::= Addressable
Identifiable ::= Container
```

Some attributes can hold a set or list of values of certain type instead of a scalar value. We use the term "*A* : {*Y*}" to specify that the attribute *A* holds a set of values of certain type *Y*. Similarly, we use the term "*A* : <*Y*>" to specify that the attribute *A* holds a list of values and the ordering of values in the list is significant. We allow the set and list to be empty unless explicitly stated otherwise. For example, the statement "*Calendar* => Invitations : {*Invitation*}" specifies that the *Calendar* has an attribute named Invitations which hold a set of *Invitation* objects. When the ordering is significant, such as the contents in an email message,

1 Since CEN is the unique identifier of the Entity, we can allow the name to be optiional.

we use the notation "*MultiContent* => Parts : <*Content*>" to specify that the Parts attribute of *MultiContent* holds an ordered list of *Content*.

We use the attribute definition modifier **immutable** as in "*A* : **immutable** *Y*" to specify that once the attribute value is set, it is final. For example, the statement "*Identifiable* => Id : **immutabe guid**" specifies that if the Id of any identifiable object is assigned at creation time, it cannot be changed afterwards.

Some attribute values of an entity can be defined to be derived by a function using the notation "*C* => *A* : *F*( ):*Y*" which states that the value of the attribute *A* of class *C* is obtained by applying function *F.* The complete definition of the function is given in a separate statement. The function *F* may take one or more arguments with the first argument being an instance of *C* (i.e. this) which is omitted in the function definition. For example, the statement *Entity* => Parent : *getParent*():*Entity* states that parent of an entity e is obtained by applying the function *getParent*(e). The definition of the function *getParent* is shown as "*getParent* : () -> *Entity*." Notice that the type of the first argument is omitted in the function definition.

We annotate the attribute definitions by the modifiers **Part**, **Ref**, or **PartRef** to specify the referential constraints on the references to entities. For example, the statement "*Calendar* => Invitations : {**Part** *Invitation*}" specifies that the Invitations attribute of *Calendar* contains a set of *Invitation* objects by strong containment relation, i.e. if the *Calendar* is deleted, all *Invitation* objects in that attribute are also deleted. When an *Invitation* object is created, the parent of the *Invitation* object must be the *Calendar* object to which it belongs.

The statement "**type** *T* **is** *S* " declares a token *T* for use in this specification as some data type *S*. Boldface names are keywords that have the customary meaning associated with its name. The statement of the form "*E* = *G* | *H*" defines an enumeration, i.e. *E* is one of *G* or *H*. For example, the statement "`Priority = High | Normal | Low` " specifies that the priority can be `High`, `Normal`, or `Low`.

2.1.1 References

In this section, we use the term *supplier* to represent a referenced object and the term *client* to represent a referencing object. Every supplier entity has a primary affinity to one client entity which is the unique parent. However, the same supplier entity can be simultaneously referenced by multiple other client entities. When an entity is referenced by multiple client entities, we need to derive the scope of the entity from the particular client entity and, inductively, the particular reference path (representing a container hierarchy) from the scope.

We use the following notations to annotate a reference type to an entity instance.

- **`Part`** *`E,`* where *E* is *Entity* or a subclass of *Entity*

  A `Part` annotation implies that there is a parent-child strong composition or contains-in relationship between the client entity and the supplier entity being referred to via the `Part` reference. The referred to object cannot exist without the referrer. In addition, this annotation implies that the client entity that refers to the supplier entity via the `Part` reference must be the unique parent of the supplier entity and they belong to the same scope.

- **`Ref`** *`E,`* where *E* is *Entity* or a subclass of *Entity*

  A `Ref` annotiation implies that the supplier entity being referenced has existence independent of the referencing client entity. A client entity can use `Ref` reference to a supplier entity which is already a target of an original `Part` reference. `Ref` reference does not enforce referential integrity. Upon the deletion of the original `Part`, the `Ref` reference will become a dangling reference. When the supplier entity is accessed via the `Ref` reference, the scope in effect is derived from the original `Part` reference.

- **`PartRef`** *`E,`* where *E* is *Entity* or a subclass of *Entity*

  A `PartRef` annotation implies that there is potentially a parent-child strong composition relationship between the client entity that uses `PartRef` reference to the supplier entity. `PartRef` reference enforces referential integrity by stipulating that more than one client entity can reference the same supplier entity, but at least one such client must exist. When there are several client entities referring to the same supplier entity through the `PartRef` references, for the purpose of binding the supplier entity to the scope and the policies within the scope, one of the client entities must be a unique parent of the supplier entity, unless there is a `Part` reference from one and only one other client entity which is invariably the parent.

Delete semantics work as follows. When a client entity is deleted, all supplier entities referred to via `Part` reference must be also deleted. A supplier entity referred to via `Ref` reference is untouched. A supplier entity referred to via a `PartRef` reference has behavior depending on whether the client entity is the unique parent of the supplier entity. If it is, then the supplier entity via a `PartRef` reference must be also deleted. If it is not, the existing parent shall prevent the cascade delete of the supplier entity.

We use the following terminology to annotate a reference to the data structure objects which are not entities but may be identiable by unique IDs:

- **`part`** *`D,`* where *D* is not *Entity*

  A `part` annotation implies that there is a 1-1 contains-in relationship between the parent object and the data structure object being referred to via the `part` relation. No other object can refer to the object that is the target of the `part` relation - it is effectively private to the container. The referred to object cannot exist without the referrer. The analogous reference for entities is annotated by `Part`.

- **`ref`** *`D,`* where *D* is not *Entity*

  A `ref` annotiation implies that the supplier object has existence independent of the client object. Identifiable objects that have globally unique ID’s can be weakly contained by this reference type. The analogous reference for entities is annotated by `Ref`.

- **`part ref`** *`D,`* where *D* is not *Entity*

  A `part ref` annotation implies that more than one client object can reference the supplier object, but at least one such client object must exist. The analogous reference for entities is annotated by `PartRef`.

The **immutable** annotation implies that the attribute is set at the object creation time and cannot be changed. The Id, CreatedOn and CreatedBy attributes are examples of immutable attributes.

2.1.2 Primitive Types

This document assumes that certain basic datatypes are understood. These datatypes are defined here.

`boolean`	Attributes of this type can be either true or false.
`boolean-Expression`	Attributes of this type hold an expression that can be evaluated to either true or false.

`date`	Attributes of this type hold a date value (year, month and day only).
`entityClass`	Attributes of this type hold a handle for a class of entities.
`eventDefinition`	Attributes of this type holds a handle of a class of events in the system.
`float`	Attributes of this type can be either integral or non-integral (real) numbers.
`groupByExpression`	Attributes of this type hold an expression that imposes the grouping of the query results.
`guid`	Attributes of this type hold globally unique values (identifiers) across all Beehive instances.
`integer`	Attributes of this type hold only whole numbers.
`locale`	Attributes of this type hold the locale names.
`octet`	Attributes of this type hold a single 8 bit value. Usually the type will be **`<octet>`** indicating a sequence of such values.
`orderByExpression`	Attributes of this type hold an expression that imposes an ordering of the query results.
`password`	Attributes of this type are meant to be kept secret and secure at the system level. Extra measures are taken to protect the privacy of the value in this attribute.
`richtext`	A sequence of Unicode characters that may include mark-up tokens. Richtext can be localized[2]. Richtext values preserve formatting such as carriage returns, colors, font modifiers, etc., and it is expected that the UI will respect this formatting.
`string`	A sequence of Unicode characters that does not include mark-up tokens. Strings can be localized[1]. It is up to the particular UI to decide how string values are rendered (contrast with richtext). String values are *always* single-line strings (i.e., no line feeds or carriage returns allowed), so they can be shown in lists, etc.
`time`	Attributes of this type hold a time value (hour, minute, second and optional further sub-second precision).
`timeoffset`	Attributes of this type hold a time period duration. Similar to SQL type INTERVAL DAY TO SECOND.

[2] Localization here means that instead of the attribute containing the actual characters, it can contain a special ID instead, where the ID refers to a locale-specific sequence of characters stored separately (e.g., a resource bundle). Localization can be *static* (defined by Oracle) or *dynamic* (defined by customer or third-party).

`timestamp`	Attributes of this type hold a date and time value, with time precision to at least milliseconds. It reflects the coordinated universal time (UTC), which is defined differently from Greenwich mean time (GMT). GMT is equivalent to universal time (UT). UTC is based on an atomic clock and UT is based on astronomical observations.
`uri`	Attributes of this type hold Universal Resource Identifiers. See Berners-Lee, T., et al. “Uniform Resource Identifiers (URI): General Syntax”, RFC 2396 [1] “Berners-Lee, T., et al. “Uniform Resource Identifiers (URI): General Syntax”, RFC 2396, August 1998.” for more details.
`viewer`	A field of this type represents the user in the current user context. In the case where the user context involves delegation, the viewer is the delegator rather than the user (delegatee) who is acting through the user context.

2.2 Conventions

The regular Times New Roman font is used for explanation of classes.

`The italic Courier New font is used for class names.`

The italic Century Gothic font in blue is used for auxiliary class names.

`The regular Courier New font is used for attribute names.`

`The regular` **`Courier New font`** `in` **`bold`** `is used for attribute modifiers.`

The regular Arial font in red is used for notes.

The italic Arial font in blue is used for issues.

3.0 Object Models

3.1 Entity

[1]
```
Identifiable =>
        Id                       : immutable guid
```

Identifiable defines the Id attribute common to all entities and to some non-entities, such as AttributeDefinition, AttributeTemplate, CategoryApplication, and LabelApplication that are not entities but need to be uniquely identified.

[2]
```
Identifiable ::= Entity
Entity =>
        [Name]                   : string,
        [Creator]                : immutable Ref Actor,
        CreatedOn                : immutable timestamp,
        ModifiedBy               : Ref Actor,
        ModifiedOn               : timestamp,
        Deleted                  : boolean,
        Parent                   : getParent():Entity,
        AttachedMarkers          : getAttachedMarkers():{Marker},
        AttachedSubscriptions : getAttachedSubscriptions():{Subscription},
        AttachedReminders        : getAttachedReminders():{Reminder},
        AttachedLabelApplications
        : getAttachedLabelApplications():{LabelApplication},
        AttachedCategoryApplications
        : getAttachedCategoryApplications():{CategoryApplication}
```

[2.1]
```
AccessControlFields =>
        ACL                      : {part ACE}
        [Owner]                  : Ref Accessor,
        OwnerGrantAccessTypes    : {part AccessType},
        OwnerDenyAccessTypes     : {part AccessType},
        Scope                    : getScope():Scope
```

An entity is a tuple with a globally unique ID and an optional name. An entity's name is mutable. Virtually all manipulable objects are entities. Access to every entity is controlled through access control fields. The owner is an accessor which can be either a single actor or a group. The actor that made the entity come into existence is the creator of the entity and this value cannot be changed. Each entity can have zero or more markers, subscriptions, reminders, label applications, and category applications associated with it.

The access control fields are a projection of the entity that includes the owner; the owner is a special accessor who can always change the security attributes of the entity that she owns. The access control fields include the access types that the owner grants or denies to herself. The entity's scope, also an access control field, refers to the scope that will be compared against the assigned scope and delegated scope of the privileges. The assigned scope must be a super scope of the entity's scope for the privileges to be asserted on the entity.

[2.2]
```
getParent : () -> Ref Entity
```

This function returns the entity that is the parent of the entity. The parent entity is determined implicitly by the primary container of the entity.

[2.3] *getAttachedMarkers* : **viewer** -> {**Ref** *Marker*}

This function computes all markers on the associated entity that are available to the viewer .

[2.4] *getAttachedSubscriptions* : **viewer** -> {**Ref** *Subscription*}

This function computes all subscriptions on the associated entity that are available to the viewer.

[2.5] *getAttachedReminders* : **viewer** -> {**Ref** *Reminder*}

This function computes all reminders on the associated entity that are available to the viewer.

[2.6] *getAttachedLabelApplications* : **viewer** -> {**part** *LabelApplication*}

This function computes the label applications assigned to the entity by the associated labels.

[2.7] *getAttachedCategoryApplications* : () -> {**part** *CategoryApplication*}

This function computes the category applications or attributions assigned to the classified entity by the associated categories.

[2.8] *getScope* : () -> **Ref** *Scope*

This function computes the scope of the entity.

3.2 Meta-Entity

3.2.1 Entity Schema

[3]
```
Entity ::= EntitySchema
EntitySchema =>
        Class                    : entityClass,
        [Description]            : richtext,
        Attributes               : {part ref AttributeDefinition}
```

An entity schema specifies the name and type of attributes in the entities of an entity class. Users can define entity schemas to extend the attributes of the entities. For example, a user can create “Legal-Document,” “PurchaseOrderDocument,” “CollegeTranscriptDocument,” etc., schemas, by creating the EntitySchema objects to extend the attributes of Document . For instance, the user can define the “CollegeTranscriptDocument” EntitySchema object that contains the attribute definitions for Student, Adviser, Year, Semester Hours, Average GPA, etc., attributes. Two or more EntitySchema objects may share the same AttributeDefinition object.

[4]
```
Identifiable ::= AttributeDefinition
AttributeDefinition =>
        Name                     : string,
        [Description]            : richtext,
        Searchable               : boolean,
        Aggregate                : boolean,
        Type                     : immutable part PropertyType,
        AllowedValues            : <part CollabPropery>,
        [DefaultValue]           : part PropertyValue,
        [MinimumValue]           : part PropertyValue,
        [MaximumValue]           : part PropertyValue,
        [MinimumValueInclusive] : boolean,
        [MaximumValueInclusive] : boolean
```

An attribute definition specifies the name, type, and enumeration of allowed values for the attributes. The type includes String, Integer, Float, Boolean, Date, etc. The definition also specifies whether the attribute is indexed for search.

[5]
```
PropertyType = String | Integer | Boolean | Float
        | Identifiable_Type | Uri_Type
        | Timestamp_Type | DateTime_Type |
        | List_Identifiable | List_String
        | List_Integer | List_Boolean
        | List_Timestamp | List_Uri | List_Float
        | List_DateTime
```

The enumeration of the type of the properties.

3.3 Core

3.3.1 Attribute and Property

[6]

```
Attribute =>
        Definition              : immutable ref AttributeDefinition,
        PropertyValue           : part PropertyValue
```

An attribute holds a property value, which is compatible with the type and constraints imposed by the attribute definition.

[7]

```
CollabProperty =>
        Name                    : string,
        [Description]           : richtext,
        Value                   : part PropertyValue
```

A collab property holds a property name and a property value. A collab property without a value does not exist in the property set. Properties in the same set must have unique names.

[8]

```
type PropertyValue is
        string | <string>
        | integer | <integer>
        | boolean | <boolean>
        | timestamp | <timestamp>
        | uri | <uri>
        | float | <float>
        | DateTime | <DateTime>
        | ref Identifiable | <ref Identifiable>
```

A property’s value can either be a value, an identifiable, a list of values, or a list of identifiables.

3.3.2 Bondable

[9]
```
Identifiable ::= Bondable
Bondable =>
        Bonds                    : getBonds():{Bond}
```

Bondable defines the aspect of an entity that can be bonded to any number (including zero) of other entities by a variety of bond types. Only entities of this type can be bonded.

[9.1]
```
getBonds : viewer -> {Ref Bond}
```

This function computes all bonds in which the associated entity is a participant.

NOTE While the ability to get all bonds connected to an entity from a bondable entity is an important part of the eventual API, its inclusion in this format should not imply a particular persistence representation. Since bonds are ultimately at least 2-way, it is appropriate to store bonds separately from any entity to which it bonds.

3.3.3 Addressable

[10]
```
Identifiable ::= Addressable
Addressable =>
        Addresses                : <part EntityAddress>,
        PrimaryAddress           : part ref EntityAddress,
        DefaultAddressForType : getDefaultAddressForType():EntityAddress,
        DefaultAddressForScheme : getDefaultAddressForScheme():EntityAddress
```

Addressable contains a set of addresses, one of which is the primary address.

[10.1]
```
getDefaultAddressForType : string -> EntityAddress
```

This function computes the default entity address for a given addess type.

[10.2]
```
getDefaultAddressForScheme : AddressScheme -> EntityAddress
```

This function computes the default entity address for a given address scheme.

[11]
```
EntityAddress =>
        AddressType              : string,
        AddressScheme            : part AddressScheme,
        Address                  : part Address
```

An entity address holds an address for an addressable. Each address is represented by a URI in one of many varieties of URI schemes. The AddressType attribute of entity address describes in a human readable way the type of the address. Some examples of address types are “Business,” “Personal,” “Assistant,” “Spouse,” etc.

[12]
```
type AddressScheme is "sip" | "mailto" | "tel" | "im"
        | "fax" | "sms" | "http" | "https" | "ldap"
        | "news" | "ftp" | "pres" | "xmpp" | "urn"
        | "orapostal" | "orapush" | "oraalert"
```

The address uri schemes are predefined literal constants. Readers are referred to http://www.iana.org/assignments/uri-schemes.html for additional uri schemes. We defined the proprietary uri scheme "orapostal" for the post office mail, "orapush" for Oracle's push protocol, and "oraalert" for mobile alerts.

[13] **type** *Address* **is uri**

There are a number of ways to address an entity; all addresses can be represented using URI syntax (see Berners-Lee, T., et al. "Uniform Resource Identifiers (URI): General Syntax", RFC 2396 [1] "Berners-Lee, T., et al. "Uniform Resource Identifiers (URI): General Syntax", RFC 2396, August 1998.").

3.3.4 Lockable

[14]
```
Identifiable ::= Lockable
Lockable =>
        Locks                   : getLocks():{Lock}
```

Lockable is an aspect of an entity that may be locked by an actor with the lock jointly held by a set of accessors. Accessors not included in this set are prevented from modifying the properties of the locked entity.

[14.1] *getLocks* : () -> {**part** *Lock*}

This function computes the locks on the associated entity.

[15]
```
Identifiable ::= Lock
Lock =>
        LockHolders             : {Ref Accessor},
        Timeout                 : timestamp,
        LockType                : part LockType,
        LockedEntity            : Ref Entity,
        [Creator]               : immutable Ref Actor,
        CreatedOn               : immutable timestamp
```

A lock is an identifiable object of a certain lock type. An actor may hold multiple locks of different types on the same entity. If the entity is a container, the lock is applied to all entities in the container.

[16] *LockType = UserRequest | Dav | All*

The enumeration of lock type, which can be user request, Dav, or all.

3.3.5 Localized String

[17]
```
LString =>
        Values                  : {part LValue}
```

An L-string is a structure that contains alternative strings for different locales.

[18]
```
LValue =>
        Locale                  : locale,
        Value                   : string
```

An L-value is a localized string for a specific locale.

3.3.6 Location

[19]
```
Location =>
        Name                    : string,
        [Description]           : richtext,
        [Mark]                  : {part PhysicalLocation},
        [TimeZone]              : Ref TimeZone
```

A location has a name and may also have an associated physical presence in space. Note that just because the physical location changes, the name need not. For example, a location name might be "On an airplane" while the physical location would be the coordinates in space of the traveler changing as the plane moves.

[20]
```
PhysicalLocation => ;
```

A physical location provides the spatial coordinates (that can be provided by GPS) or the specification of geospatial regions.

[21]
```
PhysicalLocation ::= PhysicalCoordinates
PhysicalCoordinates =>
        Latitude                : float,
        Longitude               : float,
        [Altitude]              : float
```

A physical coordinates instance is a physical location represented by the GPS coordinates of an object in space.

[22]
```
PhysicalLocation ::= OtherPhysicalSpecification
OtherPhysicalSpecification =>
        Properties              : {part CollabProperty}
```

An other physical specification uses the properties to specify an Oracle Spatial location mark or other geospatial region descriptions such as the street address, Washington DC, London, Tokyo, etc.

3.3.7 Date Time

[23]
```
DateTime =>
        Time                    : timestamp,
        [ScheduledTimeZone]     : Ref TimeZone,
        DateOnly                : boolean,
        FloatingTime            : boolean
```

A date time is used to hold a timestamp with various granularities. Optionally, the timestamp can be placed in a specific timezone.

[24]
```
DateTimeRecurrenceSet =>
        [StartDateTime]         : part DateTime,
        [InclusionRule]         : part DateTimeRecurrenceRule,
        IncludeDateTimes        : {part DateTime},
        ExcludeDateTimes        : {part DateTime}
```

A date time recurrence set is a complete set of recurrence instances for a periodic event. The date time recurrence set is generated by considering the initial StartDateTime attribute along with the InclusionRule, IncludeDateTimes, and ExcludeDateTimes attributes. The StartDateTime attribute defines the first instance in the recurrence set. The final recurrence set is generated by gathering all of the start date-times generated by the specified InclusionRule and IncludeDateTimes attribute values, and then excluding any start date-times which fall within the union of start date-times generated by any specified ExcludeDateTimes attribute values. This implies that start date-times within exclusion related entries take precedence over those specified by inclusion related entries. Where duplicate instances are generated by the InclusionRule and IncludeDateTimes attribute values, only one recurrence is considered. Duplicate instances are ignored.

[25] **`type`** *`DateTimeRecurrenceRule`* **`is string`**

A date time recurrence rule is a structured string as defined by iCalendar RFC 2445.

3.3.8 Enumerations

[26] *`Priority = High | Normal | Low`*

The enumeration of priorities for email message, voice message, occurrence, and task.

[27]
```
ProvisioningStatus = Enabled | Locked | Disabled
        | MarkedForDelete | DeleteInProgress
```

The enumeration of provisioning status of an entity, such as user, internal resource, external person, device, etc.

3.4 Container and Scope

3.4.1 Container

[28]
```
Identifiable ::= Container
Container =>
        Name                       : string,
        [ActivePreferenceProfile]: Part PreferenceProfile,
        Properties                 : {part CollabProperty},
        AttachedWorkflows          : getAttachedWorkflows():{Workflow},
        [VersionControlConfiguration]: part VersionControlConfiguration,
        [CategoryConfiguration] : part CategoryConfiguration,
        AvailableTemplates         : getAvailableTemplates():{Template},
        [WorkflowTemplate]         : getWorkflowTemplate():WorkflowTemplate,
        ChangeSummary              : getChangeSummary():{ChangeSummaryRecord}
```

Container defines a logical region or scope for policies, workflows, preferences, version control configuration, and category configurations. The available templates include the templates configured and bound to the container. Scopes and folders are containers.

[28.1] `getAttachedWorkflows : () -> {Ref Workflow}`

This function computes the workflows currently active in the container.

[28.2] `getAvailableTemplates : () -> {Ref Template}`

This function computes the templates to use when creating new entities in the container.

[28.3] `getWorkflowTemplate : eventDefinition -> Ref WorkflowTemplate`

This function computes the workflow template bound to a class of events in the container. The system shall spawn a workflow from the workflow template when such an event occurs.

[28.4] `getChangeSummary : viewer -> {part ChangeSummaryRecord}`

This function computes the change summary information for a container. This will be a collection of records containing the number of new and unread artifacts in the container, with one record for each type of artifact in the container.

[29]
```
ChangeSummaryRecord =>
        EntityClass                : entityClass,
        NewCount                   : integer,
        UnreadCount                : integer
```

A change summary record contains the number of new and unread artifacts of an entity class in the container.

[30]
```
Identifiable ::= VersionControlConfiguration
VersionControlConfiguration =>
        VersionControlModel        : part VersionControlModelType,
        MaximumVersionsToKeep      : integer,
        AutoLabel                  : boolean,
        LabelFormat              : part LabelFormatType,
        Final                    : boolean
```

A version control configuration specifies, for an associated container, whether to version the artifacts automatically or manually. It can also disable versioning of the artifacts in the container. The system may automatically purge the versions when the number of versions of an artifact exceeds the maximum versions to keep, unless the versions are marked not to auto-purge.

[31]
```
VersionControlModelType = Disabled | Auto | Manual
```

The enumeration of version control model type. Versioning of the artifacts in the container may be automatic or manual. The versioning can be also disabled.

[32]
```
LabelFormatType = IntegerFormat | LowercaseFormat
        | UppercaseFormat | RomanNumeralFormat
        | DecimalFormat
```

The enumeration of label format type. The format for version labels may be integers, lowercase or uppercase strings, Roman numerals, or decimals.

[33]
```
Identifiable ::= CategoryConfiguration
CategoryConfiguration =>
        Name                     : string,
        [Description]            : richtext,
        Templates                : {part ref CategoryApplicationTemplate},
        Final                    : boolean,
        Mandatory                : boolean,
        Parent                   : getParent():Container
```

A category configuration, which is defined on the container, configures and binds the set of category application templates to the container. If the template configuration is final, the sub-containers must use the templates as is, i.e. sub-containers cannot override the current configuration. If the Mandatory attribute of the category configuration is true, all designated entities in the container and sub-containers are required to bind by these templates.

3.4.2 Scope

[34]
```
Entity ::= Scope
Bondable ::= Scope
Addressable ::= Scope
Container ::= Scope
Scope =>
        [Template]                : Ref Template,
        [Description]             : richtext,
        RoleDefinitions           : {PartRef RoleDefinition},
        AssignedRoles             : {PartRef AssignedRole},
        Groups                    : {PartRef Group},
        AvailableMarkers          : {PartRef Marker},
        [QuotaConfiguration]      : part QuotaConfiguration
```

A scope is a logical region or neighborhood in the universe of entities. The scope contains the roles and groups. The scope also contains the library of markers. As a container, the scope can contain templates, workflows, etc., to define the administrative boundary.

[35]
```
Scope ::= Community
Community =>
        Actors                    : {PartRef Actor},
        Agents                    : {PartRef Agent},
        Organizations             : {PartRef Organization},
        Workspaces                : {PartRef Workspace},
        [QuotaConfiguration]      : part CommunityQuotaConfiguration
```

A community is a set of actors and agents that share a common set of workspaces and are governed by a common set of policies.

[36]
```
Community ::= Enterprise
Enterprise =>
        [Parent]                  : getParent():<>,
        EntitySchemas             : {Part EntitySchema},
        TimeZones                 : {Part TimeZone},
        TimeZoneAliasMaps         : {Part TimeZoneAliasMap},
        Archive                   : Part Archive
```

An enterprise is the top-level scope for all entities. For a small installation, the enterprise can represent the single community of actors, agents, and workspaces.

It also contains the metadata objects such as entity schemas, timezones, timezone alias maps, etc.

[37]
```
Community ::= Organization
Organization =>
        Parent                    : getParent():Community
```

An organization is a sub-space under an enterprise. Organizations can be nested to form hierarchies. Administrative policies associated with parent organizations flow down into child organizations and workspaces.

The qualified name of an organization is the name of itself (from the Name attribute of entity) prepended to the qualified name of its parent (separated by a dot).

[38]
```
QuotaConfiguration =>
        StorageSpaceHardQuota   : integer
```

A quota configuration specifies the hard quota for a container to impose the total amount of space that can be allocated within this container.

[39]
```
QuotaConfiguration ::= CommunityQuotaConfiguration
CommunityQuotaConfiguration =>
        DefaultStorageSpaceHardQuotaForSubOrganizations: integer,
        DefaultPersonalWorkspaceQuotaConfiguration
                                : part WorkspaceQuotaConfiguration,
        DefaultTeamWorkspaceQuotaConfiguration
                                : part WorkspaceQuotaConfiguration,
        MaximumNumberOfUsers    : getMaximumNumberOfUsers():integer,
        MaximumNumberOfWorkspaces: getMaximumNumberOfWorkspaces():integer
```

A community quota configuration specifies the quotas for sub-containers of a community. If a community administrator wants to change the quota configuration of the associated container, the change will be allowed only if it does not invalidate the sub-container settings.

[39.1]
```
getMaximumNumberOfUsers : () -> integer
```

This function computes the maximum number of users that can be created in the community. The maximum number of users is configured from the system administration console.

[39.2]
```
getMaximumNumberOfWorkspaces : () -> integer
```

This function computes the maximum number of workspaces that can be created in the community. The maximum number of workspaces is configured from the system administration console.

[40]
```
QuotaConfiguration ::= WorkspaceQuotaConfiguration
WorkspaceQuotaConfiguration =>
        StorageSpaceSoftQuota   : integer
```

A workspace quota configuration specifies the soft quota warning levels. When the soft quota is breached, a warning event will be raised. Once hard quota is reached, no more create or update (with-more-bytes) operations will be allowed until the quota breach is resolved. Since the deletion of artifacts in workspace only moves the artifacts to workspace trash, the deleted artifacts still count against quota until archived or expunged. If no workspace quota configuration is specified at the time the workspace is created, the default configuration will be inherited from the community quota configuration of the parent community.

3.5 Directory

3.5.1 User Directory

[41]
```
Entity ::= BaseAccessor
Bondable ::= BaseAccessor
BaseAccessor =>
        Parent                  : getParent():Scope,
        Groups                  : getGroups():{Group},
        AssignedRoles           : getAssignedRoles():{AssignedRole},
        EffectiveGroups         : getEffectiveGroups():{Group}
```

A base accessor can be assigned privileges via ACE's (access control entries) and assigned roles.

[41.1] `getGroups : () -> {Ref Group}`

This function computes all the groups, including the static and dynamic groups, that the base accessor is directly assigned to. This is contrasted with the function getEffectiveGroup which includes all the groups inherited through the nested groups.

[41.2] `getAssignedRoles : () -> {Ref AssignedRole}`

This function computes all the roles that the accessor is directly assigned to.

[41.3] `getEffectiveGroups : () -> {Ref Group}`

This function computes all the groups in which the associated accessor is a member of either directly or via any of the nested subgroups.

[42]
```
BaseAccessor ::= Accessor
Addressable ::= Accessor
Accessor =>
        Memberships             : {Ref Community},
        Properties              : {part CollabProperty},
        AvailablePreferenceProfiles: {Part PreferenceProfile},
        [ActivePreferenceProfile]: Ref PreferenceProfile
```

An accessor is a member of a community. An accessor can be placed in multiple communities besides the primary community. Some instances of accessor, such as users, cannot be created under the scope of a workspace; only roles and groups can be created under the scope of a workspace. An accessor may maintain a set of preference profiles, but only one profile can be active at a time.

[43]
```
Accessor ::= Actor
Actor =>
        Parent                  : getParent():Community,
        Principals              : {Part Principal},
        Status                  : part ProvisioningStatus
```

An actor is an entity that can act on the system. Every actor must belong to one primary community, either an enterprise or organization, and can optionally belong to multiple secondary organizations. Each actor may contain a list of principals. Only one principal can be active at a time for authorization purposes. An actor may use a "delegated" principal that lets the actor perform certain actions on behalf of another actor.

[44]
```
Addressable ::= Person
Person =>
```

```
        [GivenName]             : string,
        [MiddleName]            : string,
        FamilyName              : string,
        [Prefix]                : string,
        [Suffix]                : string,
        [Nickname]              : string,
        [JobTitle]              : string,
        [Department]            : string,
        [OfficeLocation]        : string,
        [Company]               : string,
        [Profession]            : string,
        [TimeZone]              : Ref TimeZone,
        [Locale]                : locale
```

Person defines the attributes of the user and external person.

[45]
```
Actor ::= User
Watchable ::= User
InstantMessageRecipient ::= User
Person ::= User
User =>
        [PersonalWorkspace]     : Ref PersonalWorkspace,
        AccessibleWorkspaces    : getAccessibleWorkspaces():{Workspace},
        FavoriteWorkspaces      : getFavoriteWorkspaces():{Workspace}
```

A user has given, middle, and family names. A user is an entity that can act upon other entities. A user is watchable and thus exhibits presence and must have at least one presence mapping. A user is also an instant message recipient, and therefore, supports the necessary attributes to retrieve the online instant messages.

[45.1] `getAccessibleWorkspaces : () -> {Ref Workspace}`

This function computes the accessible workspaces of the user.

[45.2] `getFavoriteWorkspaces : () -> {Ref Workspace}`

This function computes the favorite workspaces of the user.

[46]
```
User ::= OrganizationUser
OrganizationUser =>
        [Manager]               : Ref OrganizationUser,
        [Assistant]             : Ref OrganizationUser,
        DirectReports           : {Ref OrganizationUser}
```

An organization user is a member of an organizational hierarchy.

[47]
```
Actor ::= SystemActor
SystemActor => ;
```

A system actor represents a privileged, non-human user of the system. It represents a software agent that performs email delivery, notification, workflow, record management, etc. System actors, because they are usually native to the system or their actions are programmed into the system, are often not authenticated.

[48]
```
Accessor ::= Group
Watchable ::= Group
```

```
Group =>
        Name                    : string,
        [Description]           : richtext,
        Members                 : getMembers():{Actor},
        MemberActors            : {Ref Actor},
        MemberAgents            : {Ref Agent},
        MemberPrincipals        : {Ref Principal},
        SubGroups               : {PartRef Group},
        [Workspace]             : Ref TeamWorkspace,
        EffectiveMemberActors   : getEffectiveMemberActors():{Actor},
        EffectiveMemberAgents   : getEffectiveMemberAgents():{Agent},
        EffectiveMemberPrincipals
        : getEffectiveMemberPrincipals():{Principal}
```

A group is addressable and is an accessor which can own entities. It contains a set of actors, principals, and sub-groups for access control. A group may include agents, such as board of directors, who cannot access the system but can communicate with the system or other actors via messages. A group must have a unique name. The members of the group are explicitly added to or deleted from the group. The super-group of a group is given by the Groups attribute defined in Accessor.

[48.1] `getMembers : () -> {Ref Actor}`

This function computes the actors who are direct members of the associated group.

[48.2] `getEffectiveMemberActors : () -> {Ref Actor}`

This function computes all the actors who are members of the associated group either directly or via any of the nested subgroups.

[48.3] `getEffectiveMemberAgents : () -> {Ref Agent}`

This function computes all the agents who are members of the associated group either directly or via any of the nested subgroups.

NOTE We believe the Agents attribute is a separate attribute because agents cannot access the system; agents can only send to or receive messages from the system. We put Principals separately in Principals attribute because Principal is not Addressable whereas Group, Accessor, and Agent are Addressable. When you address the group, you are addressing all the accessors and agents in the group.

[49]

```
Group ::= DynamicGroup
DynamicGroup =>
        Query                   : booleanExpression,
        ExcludedMemberActors    : {Ref Actor},
        Members                 : getMembers():{Actor}
```

A dynamic group contains a list of members and agents that satisfy the query criteria. A dynamic group can also contain members and agents who are explicitly added to or excluded from the group.

Actors can opt out of a group, especially from the groups that they belong to by super-group inheritance, without affecting their privileges. When an actor opts out, the actor is placed in the exclusions set that excludes them from the addressable members of the groups. The actors in the exclusion set will still derive the grant and deny privileges from the group, if they belong to the group directly or via a sub-group.

[49.1] *getMembers* : () -> {**Ref** *Actor*}

This function computes the actors who are direct members of the associated dynamic group. It is the result of substracting the excluded member actors from the member actors and adding the actors from the query.

[50]
```
Entity ::= Agent
Bondable ::= Agent
Addressable ::= Agent
Agent =>
        Parent                  : getParent():Community,
        Memberships             : {Ref Community},
        Groups                  : {Ref Group},
        Status                  : part ProvisioningStatus
```

An agent is an external person or resource with whom the actors in the system need to collaborate through messages.

[51]
```
Agent ::= ExternalPerson
Person ::= ExternalPerson
ExternalPerson =>
        Properties              : {part CollabProperty}
```

An external person can merely interact with the system via messages (e.g., can send a message into the system or receive a message from the system).

3.5.2 Resource Directory

[52]
```
Addressable ::= Resource
Resource =>
        Name                      : string,
        [Description]             : richtext,
        [StationaryLocation]      : part Location
```

Resource defines the attributes, such as the stationary location attribute, common to internal and external resources.

[53]
```
Actor ::= InternalResource
Resource ::= InternalResource
InternalResource =>
        [ResourceWorkspace]       : Ref Workspace,
        [Identifier]              : string
```

An internal resource is a non-user actor within the system. A delegate/contact person for the internal resource could act on a system and perform operations on other entities on behalf of the resource. The internal resource can own artifacts. It has privileges and preferences that can be delegated to a user. For these reasons, an internal resource must be an actor and cannot be merely an agent. An internal resource can be directly involved in an occurrence, such as conferences or meetings, as a participant. Contrast this to the artifacts that can be merely attached or bonded to an occurrence.

[54]
```
InternalResource ::= BookableResource
BookableResource =>
        Type                      : immutable part BookableResourceType,
        [Capacity]                : integer,
        [BookingInfo]             : richtext,
        Approvers                 : {Ref OrganizationUser}
```

A bookable resource can be included in a calendar event. By doing so, the resource will be marked as reserved for the timeslot. A meeting within the resource calendar is typically initiated by a user.

NOTE Approvers attribute of BookableResource may be superceded by the coordinator role of the resource's workspace when we refine the concept of the resource workspace and workspace coordinator role.

[54.1]
```
BookableResourceType = Room | Equipment | Other
```

The enumeration of bookable resource type.

[55]
```
Agent ::= ExternalResource
Resource ::= ExternalResource
ExternalResource =>
        [Capacity]                : integer
```

An external resource is an agent outside the control of the system. It does not interact with the system but can only exchange messages. It can be used as a participant in meetings. It serves as an address holder for contacting the external person representing the resource. An external resource may have a stationary location, which in turn provides the time zone.

The external resources, for example the conference rooms in the Moscone Center, may participate in the OpenWorld events. For such an external resource, Moscone Center will be a contact address.

3.5.3 Address Book

[56]
```
Folder ::= AddressBook
AddressBook =>
        Entries                 : {Part AddressBookElement},
        SubAddressBooks         : {Part AddressBook},
        PersonContacts          : getPersonContacts():<PersonContact>,
        GroupContacts           : getGroupContacts():<GroupContact>,
        ResourceContacts        : getResourceContacts():<ResourceContact>
```

An address book is a folder that contains bookmarks to addressable entities in the user and resource directories as well as personal contact entries. Address books may contain sub-books (chapters) that can be used to hierarchically organize and share contacts.

[56.1] `getPersonContacts : () -> <Ref PersonContact>`

This function computes the person contacts from among the addresss book elements.

[56.2] `getGroupContacts : () -> <Ref GroupContact>`

This function computes the group contacts from among the addresss book elements.

[56.3] `getResourceContacts : () -> <Ref ResourceContact>`

This function computes the resource contacts from among the addresss book elements.

NOTE An address book can be owned by a group of actors.

[57]
```
Artifact ::= AddressBookElement
Addressable ::= AddressBookElement
AttachmentHolder ::= AddressBookElement
AddressBookElement =>
        [SpeedDial]             : getSpeedDial():integer,
        [Priority]              : getPriority():Priority,
        [PeopleList]            : part PeopleList,
        GroupContacts           : getGroupContacts():{GroupContact}
```

An address book element is an artifact in the address book.

[57.1] `getSpeedDial : viewer -> integer`

This function computes the speed dial number which is a unique, viewer private property representing the default telephone number of the address book element.

[57.2] `getPriority : viewer -> Priority`

This function computes the priority which is a unique, viewer private property.

[57.3] `getGroupContacts : () -> {Ref GroupContact}`

This function computes the group contacts that contain the associated element.

[57.4] `PeopleList => ;`

An address book element marked by PeopleList tag will be part of a short list of contacts with whom the user regularly interacts with via email, voice, conference, chat, etc.

[57.5]
```
PeopleList ::= BuddyList
BuddyList => ;
```

An address book element marked by BuddyList tag will be part of a shorter list of contacts that the user can observe presence and interact with via instant messages. The BuddyList contacts must have instant message addresses in order for the contacts to appear in the instant message roster. A BuddyL-ist contacts is a subset of the PeopleList contacts.

[58]
```
AddressBookElement ::= PersonContact
Person ::= PersonContact
PersonContact =>
        [Bookmark]              : Ref Person,
        [FamilyName]            : string
```

A person contact is an entry in the address book that contains the contact information for a person. It can be used to bookmark a person in the community. Alternatively, it can hold all the attributes about a person without bookmarking a person in the community.

[59]
```
AddressBookElement ::= ResourceContact
Resource ::= ResourceContact
ResourceContact =>
        [Bookmark]              : Ref Resource,
        [Capacity]              : integer,
        [ResourceName]          : string
```

A resource contact is an entry in the address book that contains the contact information for a resource. It can be used to bookmark a resource in the community. Alternatively, it can hold all the attributes about a resource without bookmarking a resource in the community.

[60]
```
AddressBookElement ::= GroupContact
GroupContact =>
        [Bookmark]              : Ref Group,
        ExternalMembers         : {part ExternalContact},
        Members                 : {Ref Accessor},
        Elements                : {Ref AddressBookElement},
        [GroupName]             : string
```

A group contact is a personal group in the address book that can optionally bookmark a group and/or include external contacts. The address book owner can include other addressable contacts in the personal group.

[61]
```
Identifiable ::= ExternalContact
ExternalContact =>
        [Name]                  : string,
        Addresses               : {part EntityAddress}
```

An external contact can be created in a group contact without having to create an external person entity in the community.

3.5.4 Presence

[62]
```
Identifiable ::= Watchable
Watchable =>
        Presence                : getPresence():Presence
```

Watchable defines the attributes common to entities that exhibit presence.

[62.1]
```
getPresence : viewer -> Presence
```

This function maps a particular viewer to a state or view of a presence.In the case where the user context involves delegation, the viewer is the delegator rather than the user (delegatee) who is acting through the user context. This lets the delegatee watch the presentity from the perspective of the delegator.

[63]
```
Entity ::= Presence
Presence =>
        Presentity              : Ref Watchable,
        Status                  : part PresenceStatus,
        [Location]              : part Location,
        [Note]                  : richtext,
        ContactMethods          : <part ContactMethod>,
        CurrentActivities       : {part Activity}
```

A presence conveys the ability, willingness, reachability, and activity of a presentity to interact or be watched by the viewer. The presented status and location may be different from the true status or location of the presentity. That is, a presentity may present status and location differently to different viewers. A presence provides a list of contact methods that describe to the viewer how to reach the presentity. The viewer can choose any one of the contact methods based on circumstances. The viewer can judge from the activity when and which method is appropriate to communicate with the presentity. If presence projection is empty (blocked) or if there is no directed presence for the viewer, the presence status will be shown as unknown but all other filelds in the presence object will be empty.

[64]
```
PresenceStatus = UserPresenceStatus | GroupPresenceStatus
```

The enumeration of presence status.

[65]
```
Presence ::= UserPresence
UserPresence =>
        Presentity              : Ref User,
        Status                  : part UserPresenceStatus
```

A user presence represents the ability, willingness, reachability, and activities of the user. From among the ordered list of directed presences in the user’s active presence configuration, the first directed presence that maps to the watcher shall be used to compose the state of the presence.

[66]
```
UserPresenceStatus = Available | NotAvailable | Unknown
        | Busy | DoNotDisturb | Away
```

The enumeration of presence status of a user. the presence status typically changes over time.

[67]
```
Entity ::= PresenceConfiguration
PresenceConfiguration => ;
```

A presence configuration contains the user settings that can be used to compose the state of the presence. It can be configured as a user or group preference and activated through the preference profiles.

[68]
```
PresenceConfiguration ::= UserPresenceConfiguration
UserPresenceConfiguration =>
        CustomPresenceProfiles  : {part CustomUserPresenceProfile},
        DirectedPresences       : <part DirectedUserPresence>
```

A user presence configuration contains the directed presences that can be used to compose the user presence. The first directed presence that applies to the watcher shall be the one used.

[69]
```
CustomUserPresenceProfile =>
        Status                  : part UserPresenceStatus,
        [Note]                  : richtext
```

A custom user presence profile specifies the status and note to be presented to the watcher.

[70]
```
DirectedUserPresence =>
        Watcher                 : Ref Accessor,
        Projection              : part Projection,
        Pending                 : boolean,
        [Status]                : part CustomUserPresenceProfile
```

A directed user presence specifies the custom user presence status and note to be presented to the watcher. If the status is not specified, the actual presence status will be derived from the current set of contact methods and activities. Since Accessor includes Actor and Group, the user presence can be directed to a group of actors as well as to an individual actor.

[71]
```
Projection = Full | Basic | Empty
```

The enumeration of projection. The projection lets the presentity direct which parts of the presence shall be shown to a particular accessor. For the user or group presence, the directed presence can specify empty projection to show the presence status as unknown and block other attributes of the presence. To show the proper presence status, either directed or actual, the basic projection must be specified. If the full projection is specified, the contact methods and activities will be shown in the user presence. For the group presence, the full projection will show the available members of the group.

[72]
```
Identifiable ::= Activity
Activity =>
        [Reference]             : Ref Entity,
        Type                    : part ActivityType,
        Realm                   : string,
        [Start]                 : timestamp,
        [End]                   : timestamp
```

An activity can be derived from the planned activities, free busy intervals, and real-time status of user sessions for conference, voice, etc. The Reference attribute indicates the source (conference, occurrence, etc) from which the activity is derived. The activity is supplied in the presence to indicate what the presentity is doing. The viewer can judge from the activity when and how is appropriate to communicate with the presentity. The realm can be used to indicate the sphere of the activity, such as official, personal, or entertainment.

[73]
```
ActivityType = OnThePhone | Conference
        | Meeting | Travel | Steering | Meal | OutOfOffice
        | Holiday | Vacation | Other
```

The enumeration of activity type. Each activity type implies availability of the presentity.

[74]
```
ContactMethod =>
        Address                 : part EntityAddress,
        [Status]                : part ReachabilityStatus,
        [ResourceId]            : string
```

A contact method describes how to reach a user. The presence contains a list of contact methods to give the viewer choices.

[75]
```
ReachabilityStatus =  Reachable | NotReachable | Unknown | Away
```

The enumeration of reachability status. The reachability status of the contact method is part of the presence information about the presentity.

3.5.5 Free Busy

[76]
```
FreeBusy =>
        Start                   : timestamp,
        End                     : timestamp,
        Intervals               : <part FreeBusyInterval>,
        CreatedOn               : timestamp
```

A free busy defines the availability of an occurrence participant or a presentity (a watchable entity) for a certain period delimited by [Start, End]. Each period of time interval is represented by a free busy interval.

[77]
```
FreeBusyInterval =>
        Type                    : part FreeBusyType,
        [Start]                 : timestamp,
        [End]                   : timestamp
```

A free busy interval indicates the free busy condition of an occurrence participant. The absence of the Start attribute represents the start time of the containing FreeBusy object; the absence of the End attribute represents the end time of the containing FreeBusy object.

[78]
```
FreeBusyType = Free | Busy | Tentative
        | OutsideAvailableHours | OutOfOffice
        | Unknown
```

The enumeration of free busy type. Each free busy type implies availability of the participant.

[79]
```
computeFreeBusy : ({Actor}, timestamp, timestamp) -> FreeBusy
```

This function computes the free busy for a time interval for a given set of actors.

3.6 Access Control

[80]
```
Entity ::= RoleDefinition
RoleDefinition =>
        [Description]           : richtext,
        Parent                  : getParent():Scope,
        Privileges              : {part Privilege},
        GrantAccessTypes        : {part AccessType},
        DenyAccessTypes         : {part AccessType},
        AlwaysEnabled           : boolean
```

A role definition is a named set of privileges and access types.

[81]
```
Entity ::= AssignedRole
AssignedRole =>
        [Description]           : richtext,
        RoleDefinition          : Ref RoleDefinition,
        Parent                  : getParent():Scope,
        AssignedScope           : Ref Scope,
        Accessors               : {Ref BaseAccessor}
```

An assigned role assigns a role definition to a set of accessors for operations within an assigned scope.

[82] **type** *Privilege* **is enum**

A privilege includes provisioning types such as EMAIL_USER, CONF_USER, etc. It can only be granted via roles.

[83] **type** *AccessType* **is enum**

Access type includes the READ, WRITE, DELETE, EXECUTE, and DISCOVER types.

[84]
```
ACE =>
        Accessor                : Ref BaseAccessor,
        GrantAccessTypes        : {part AccessType},
        DenyAccessTypes         : {part AccessType}
```

An access control entry (ACE) specifies what privileges should be granted to or denied from the given accessor. Typically an ACE is an entry in the access control list (ACL) associated with an entity or sensitivity to control access to the entity. An ACE can only be added, deleted, or modified by an actor with MODIFY_ACL privileges.

[85]
```
BaseAccessor ::= Principal
Principal =>
        Actor                   : getActor():Actor
```

A principal represents an actor for authorization purposes. A principal can correspond to one of many login ids, personal identification numbers, passwords, biometrics, or certificate credentials of an actor. Usually the actor authenticates necessary credentials in order to activate a principal.

3.7 Participant

[86]
```
Participant =>
        [Participant]           : Ref Addressable,
        [Address]               : part Address,
        [Name]                  : string
```

A participant represents various modes of participation of any addressable entity in collaboration activities, such as meetings, tasks, conferences, discussions, and messaging. If the Participant attribute is not specified, the Address attribute must be specified. An address without the addressable represents an external participant. If both addressable and address are specified, the address may be one of the addresses of the addressable or may be a private address of the addressable known only to the organizer of activities.

[87]
```
Participant ::= UnifiedMessageParticipant
UnifiedMessageParticipant =>
        [FullAddress]           : string,
        [LocalPart]             : string,
        [DomainPart]            : string,
        [DisplayPart]           : <octet>,
        [DisplayCharacterSet]   : string
```

A unified message participant represents someone involved in messaging activity or someone organizing the occurrences and todos but is herself not an occurrence or todo participant. Full address is of the form “Smith, John” <jsmith@acm.org>. If the full address is not specified, then it can be composed from “DisplayPart” <LocalPart@DomainPart>. The display part represents the array of bytes for any specified character set that are not converted into unicode or UTF16. If the Participant attribute is specified in terms of an addressable entity, then all of these address parts are optional.

[88]
```
Participant ::= ConferenceParticipant
ConferenceParticipant =>
        [Principal]             : Ref Principal,
        [Key]                   : string,
        Properties              : {part CollabProperty}
```

A conference participant is a a participant in a conference. If the participant is an actor, it holds the self or delegated principal of the actor used for authentication. If the participant is an agent, it holds the key for the agent to sign on to the system. The key is used because the agent cannot authenticate with the system the way an actor can authenticate with the system.

[89]
```
Participant ::= OccurrenceParticipant
OccurrenceParticipant =>
        [Role]                  : part ParticipantRole,
        ParticipantStatus       : part OccurrenceParticipantStatus,
        Properties              : {part CollabProperty},
        DirectlyInvited         : boolean,
        RSVP                    : boolean,
        InvitationDeliveryStatus: part ParticipantDeliveryStatus,
        DeliveryChannels        : <part AddressScheme>
```

An occurrence participant holds the status, role, and private properties of individual participants. The DirectlyInvited attribute allows for the distinction of direct invitation and invitation via group membership or delegation.

[90]
```
Participant ::= OccurrenceCompositeParticipant
OccurrenceCompositeParticipant =>
        [Role]                 : part ParticipantRole
```

An occurrence composite participant is used to invite members of a group or a workspace.

[91]
```
OccurrenceCompositeParticipant ::= OccurrenceWorkspaceParticipant
OccurrenceWorkspaceParticipant =>
        InviteEnrollees        : boolean
```

An occurrence workspace participant is used to add an invitation to the workspace default calendar. The attribute InviteEnrollees specifies whether the enrollees should be invited to the meeting.

[92]
```
OccurrenceCompositeParticipant ::= OccurrenceGroupParticipant
OccurrenceGroupParticipant => ;
```

An occurrence group participant is used to invite a group in the community or a group contact in the address book.

[93]
```
Participant ::= TodoParticipant
TodoParticipant =>
        [Role]                 : part ParticipantRole,
        ParticipantStatus      : part TodoParticipantStatus,
        Properties             : {part CollabProperty},
        DirectlyInvited        : boolean,
        RSVP                   : boolean,
        [PercentComplete]      : integer,
        [TimeAllocated]        : timeoffset,
        [TimeSpent]            : timeoffset,
        [Mileage]              : string,
        [BillingInfo]          : string,
        [Comment]              : string,
        AssignmentDeliveryStatus: part ParticipantDeliveryStatus,
        DeliveryChannels       : <part AddressScheme>
```

A todo participant holds the status, role, optional time, and private properties of individual participants. The DirectlyInvited attribute allows for the distinction of direct invitation and invitation via group membership or delegation. The PercentComplete attribute is similar to the Status attribute. It is read-only for internal participants and mapped to the value of the AssigneePercentComplete attribute in the associated assignment. For external participants, the value can be adjusted by the assigner. The assigner can supply an optional comment.

[94]
```
ParticipantDeliveryStatus = Delivered | Pending | Failed
```

The enumeration of participant delivery status.

[95]
```
Participant ::= TodoCompositeParticipant
TodoCompositeParticipant =>
        [Role]                 : part ParticipantRole
```

A todo composite participant is used to assign tasks to a group of participants.

[96]
```
TodoCompositeParticipant ::= TodoGroupParticipant
TodoGroupParticipant => ;
```

A todo group participant is used to assign a task to a group in the community or a group contact in the address book.

[97]
```
ParticipantRole = RequiredParticipant
        | OptionalParticipant
        | NonParticipant
        | Chair
```

The enumeration of participant role.

[98]
```
Participant ::= WorkspaceParticipant
WorkspaceParticipant =>
        Participant             : Ref Accessor,
        AssignedRoles           : getParticipantRoles():{AssignedRole},
        Properties              : {part CollabProperty}
```

A workspace participant defines the role of an accessor in a team workspace. The accessor can be a member (full participant), a coordinator, or a viewer.

[98.1]
```
getParticipantRoles : () -> {Ref AssignedRole}
```

This function computes the roles assigned to the accessor by the associated workspace.

3.8 Artifact Management

3.8.1 Artifact

[99]
```
Entity ::= Artifact
Bondable ::= Artifact
Artifact =>
        [UserCreatedOn]          : timestamp,
        [UserModifiedOn]         : timestamp,
        Properties               : {part CollabProperty},
        ViewerProperties         : privateProperties():{CollabProperty},
        Unread                   : isUnread():boolean,
        New                      : isNew():boolean,
        Recent                   : isRecent():boolean
```

An artifact is a type of bondable entity that comes into existence at a well-defined time and is operated upon by actors. The UserCreatedOn and UserModifiedOn attributes hold the times when the artifact is created or modified offline; these time stamps may be different from the times in the system's CreatedOn and ModifiedOn attributes.

Associated with each artifact are a set of properties that are private to each viewer. Every artifact has a set of attributes such as the date and time the artifact was last modified and by whom and a set of shared properties.

[99.1] `privateProperties : (viewer, [ArtifactContainer]) -> {CollabProperty}`

This function computes the viewer's own set of properties from the purview of an artifact container.

[99.2] `isNew : Actor -> boolean`

This function, associated with an artifact, determines for a given actor whether the artifact is considered "new". A new artifact is one that has come into existence since the last time this actor viewed the artifact's folder.

[99.3] `isUnread : Actor -> boolean`

This function, associated with an artifact, determines for a given actor whether the artifact is considered "unread". An unread artifact is one that has been modified (from the viewing actor's perspective) since the last time the actor viewed the artifact or it is one that has been created since the last time the actor viewed the artifact's folder. Note that if an artifact is new, it is also unread.

Moving an artifact does not alter its unread status. Copying an artifact also copies the unread status. Modifying viewer properties does not cause the artifact to become unread for any other actor (since other actors cannot see those changes).

We may add a private (per actor) property that allows an actor to manually set the unread flag that would override the result of the `isUnread()` function.

[99.4] `isRecent : (Actor, timeoffset) -> boolean`

This function, associated with an artifact, determines for a given actor whether the artifact is considered "recent". A recent artifact is one that has been created or modified within the past specified time interval, regardless of whether the actor has viewed the artifact in that time period. This allows an actor to keep track of changing artifacts based purely on time.

[100]
```
Sizable ::= Compressable
Compressable =>
        Name                   : string,
        ModifiedOn             : timestamp
```

Compressable is an aspect of an artifact, e.g. document, that can be compressed.

[101]
```
EntitySchema ::= DocumentSchema
DocumentSchema => ;
```

A document schema contains the attribute definitions for different types of documents. It also holds metadata and connection information about a family of documents in external repositories.

[102]
```
BasicTemplate ::= DocumentTemplate
DocumentTemplate =>
        [EntitySchema]         : Ref DocumentSchema
```

A document template captures the forms and default seetings for the documents.

[103]
```
Artifact ::= Document
Sizable ::= Document
Versionable ::= Document
Attachable ::= Document
VirusScannable ::= Document
Lockable ::= Document
Compressable ::= Document
Document =>
        [Template]             : Ref DocumentTemplate,
        Content                : part Content,
        [Parts]                : part MultiPart,
        TotalSize              : getTotalSize():integer
```

A document is an artifact that can contain single or composite contents for any assortment of media types. A more complex type of document can contain one or more component artifacts.

[103.1]
```
getTotalSize : () -> integer
```

This function computes the aggregate size of the document and all of its versions.

[104]
```
MultiPart =>
        Parts                  : <PartRef Artifact>,
        [MediaType]            : part MIME-Multipart-Type
```

A multi-part instance represents the composite parts of a document. The Parts attribute specifies the artifact components. Whether the ordering of the parts is significant depends on the multipart type.

The multi-part can model various multipart MIME encapsulations. The multi-part type must be set to `multipart/mixed`, `multipart/alternative`, `multipart/related`, `multipart/parallel`, `multipart/signed`, `multipart/encrypted`, `multipart/digest`, etc.

[105]
```
Artifact ::= Link
Link =>
        Reference              : Ref Artifact
```

A link is a symbolic link to an artifact for the purpose of including the same artifact in more than one folder or workspace. Privileges on the referenced artifact are inherited only from the primary folder or workspace of the referenced artifact (not from the link or container of the link).

[106]
```
Artifact ::= ExternalArtifact
ExternalArtifact =>
        Location                : uri,
        MediaType               : part MIME-Content-Type,
        [ContentEncoding]       : string
```

An external artifact is an artifact that is located outside of the system. We simply record its location (e.g., a uri) instead of the content itself.

[107]
```
Artifact ::= WikiPage
Sizable ::= WikiPage
Versionable ::= WikiPage
Lockable ::= WikiPage
Document =>
        [Description]           : richtext,
        Content                 : part Content,
        TotalSize               : getTotalSize():integer,
        ViewCount               : getViewCount():integer,
        Path                    : getPath():string
```

A wiki page is an artifact that is continuingly being revised. The revisions are always versioned and maintained. A wiki page holds the “raw” form of the content which often includes mark-ups using some wiki syntax, such as Creole. An implementation can render the content into an HTML representation.

3.8.2 Artifact Container

[108]
```
Container ::= ArtifactContainer
ArtifactContainer =>
        Elements                : getElements():{Artifact}
```

ArtifactContainer defines the interface common to workspace and folder.

[108.1]
```
getElements : () -> {Part Artifact}
```

This function computes the contents of the artifact container.

[109]
```
EntitySchema ::= FolderSchema
FolderSchema => ;
```

A folder schema contains the metadata and in some cases connection information about a family of folders in external repositories.

[110]
```
BasicTemplate ::= FolderTemplate
FolderTemplate =>
        [EntitySchema]          : Ref FolderSchema
```

A folder template captures the forms and default settings for the folders.

[111]
```
Artifact ::=Folder
ArtifactContainer ::= Folder
Folder =>
        [Template]              : Ref FolderTemplate,
        [Parent]                : getParent():ArtifactContainer,
        [Description]           : richtext
```

A folder is a special class of artifact in that its main purpose is to contain other artifacts. Every folder except the root folder will have exactly one parent folder. The parent of the root folder will be a workspace. Subclasses of folder should enforce their own semantics on elements of the folder.

Every artifact must be contained in exactly one folder (primary affinity). However, another folder may refer (or link) to the same artifact (secondary affinity). A folder can be shared among different workspaces. At any time, there's one primary container for a folder. Folder level policies must be compatible with the policies of the primary container.

Before a folder is created under a scope or moved from one scope to another scope, the system must check that the folder policies are compatible with the policies of the destination scope. There are two compatibility criteria:

A folder cannot define policies that conflict with those of its primary container.

Policies at the folder can extend those defined at the destination container scope.

Each subclass of folder must implement the function getElements() which returns the contents of the folder.

NOTE The folder may hold the contained entities in any attribute. The subclasses of the folder (such as the Calendar) can classify the entities under different attributes (such as invitations and occurrences attributes). Thus the Elements in a Calendar object are the union of Occurrences and Invitations.

[112]
```
Folder ::= VirtualFolder
VirtualFolder =>
        Sources                 : {part View}
```

A virtual folder does not contain its elements directly. Instead, it is the union of various folder sources (views).

[113]
```
View =>
        BaseContainer           : Ref Container,
        [Filter]                : booleanExpression
```

A view is a subset of elements from some container, filtered by an expression. The expression is applied to each element of the base container and if it evaluates to true, the element is included in the view. If the filter is absent, then that is equivalent to the **`booleanExpression`** returning **`true`** which means that all elements in the base folder or scope are in the view.

[114]
```
VirtualFolder ::= SearchResult
SearchResult =>
        Entities                : getEntities():<Entity>,
        NumberOfHits            : getNumberOfHits():integer,
        ElementProperties       : getElementProperties():{CollabProperty}
```

A search result contains the artifacts, whch match the search query, ordered and grouped by the specified criteria.

[114.1] `getEntities : () -> <Ref Entity>`

This function computes the list of entities ordered and grouped according to the specified criteria.

[114.2] `getNumberOfHits : () -> integer`

This function computes the estimated number of hits for the search query.

[114.3] `getResultItemProperties : Entity -> {part CollabProperty}`

This function computes the entity specific properties, such as the relevancy score, snippet (keywords, summary), info source, data group, federation id, etc., from the associated search result.

[115]
```
View ::= SearchView
SearchView =>
        [OrderBy]               : orderByExpression,
        [GroupBy]               : groupByExpression,
        Count                   : integer
```

A search view specifies the order by expression and group by expression.

[116]
```
Folder ::= Trash
Trash =>
        Elements                : getElements():{Artifact},
        TrashItems              : getTrashItems():{TrashItem}
```

A trash is a special container associated with a workspace to hold the trash items of the workspace.

[116.1] `getElements : () -> {Part Artifact}`

This function computes the trash artifacts in the trash container.

[116.2] *getTrashItems* : () -> {**part ref** *TrashItem*}

This function computes the trash items associated with the trash artifacts.

[117]
```
Identifiable ::= TrashItem
TrashItem =>
        DeletedOn                : timestamp,
        DeletedBy                : Ref Actor,
        OriginalName             : string,
        OriginalParent           : Ref Entity,
        DeletedEntity            : Part Entity
```

A trash item captures information about deleted entity in the workspace trash folder. It is a first-class artifact used to maintain the snapshot of the extended ACL of the entity at the time the entity was deleted. This way, the entity can be restored by the authorized actor and also its ACL can be restored.

[118]
```
Folder ::= Archive
Archive =>
        Elements                 : getElements():{Artifact},
        ArchiveItems             : getArchiveItems():{ArchiveItem}
```

An archive contains the entities that are being purged from the enterprise. Administrators can restore the accidentally purged entities. Enterprise preferences include settings for archiving.

[118.1] *getElements* : () -> {**Part** *Artifact*}

This function computes the archived artifacts in the archive folder.

[118.2] *getArchiveItems* : () -> {**part ref** *ArchiveItem*}

This function computes the archive items associated with the archived artifacts.

[119]
```
Identifiable ::= ArchiveItem
ArchiveItem =>
        ArchivedOn               : timestamp,
        ArchivedBy               : Ref Actor,
        OriginalName             : string,
        OriginalParent           : Ref Entity,
        OriginalWorkspace        : Ref Workspace,
        ArchivedEntity           : Part Entity,
        ArchivedEntitySize       : integer
```

An archive item captures information about deleted entity in the enterprise archive folder. It is a first-class artifact used to maintain the snapshot of the extended ACL of the entity at the time the entity was deleted. This way, the entity can be restored by the authorized actor and also its ACL can be restored.

3.8.3 Artifact Version

[120]
```
Identifiable ::= Versionable
Versionable =>
        Version                : PartRef Version,
        WorkingCopy            : boolean,
        VersionControlled      : boolean,
        Family                 : boolean,
        Versioned              : boolean,
        CheckedOut             : boolean,
        CheckedOutBy           : Ref Actor,
        CheckedOutOn           : timestamp,
        CheckoutComments       : string,
        CurrentVersion         : representativeVersion():Version,
        VersionHistory         : getVersionHistory():<Version>
```

Versionable defines the attributes of a versionable artifact. A versionable artifact represents a specific "frozen version" of an artifact. It holds a version node that contains the version number, label, and description. The container of the versionable artifact holds a constant EID whose representative version varies depending on the individual actor and environment. When the representative version varies, the artifact's version number, label, description, and content will vary. Starting from the representative version under the container, the actor can get to the version history and traverse the artifact version nodes. Each artifact version node holds the specific "fozen version" of the artifact.

[120.1] *representativeVersion* : (*Actor*, *Environment*) -> **PartRef** *Version*

For a given artifact, the representative version function computes the proper artifact version for the actor in the specified environment.

[120.2] *getVersionHistory* : (*Actor*, *Environment*) -> <**PartRef** *Version*>

For a given artifact, the version history function computes the version history for the actor in the specified environment.

[121]
```
Entity ::= Version
Bondable ::= Version
Version =>
        [Description]          : richtext,
        ParentArtifact         : PartRef Versionable,
        VersionArtifact        : PartRef Versionable,
        Predecessor            : PartRef Version,
        Successors             : <PartRef Version>,
        AutoPurge              : boolean,
        Finalized              : boolean,
        VersionNumber          : integer,
        [VersionLabel]         : string
```

A version represents a node in the version history. The predecessor and successor attributes of each node link up the nodes in the version history into a directed acyclic graph. The version node holds a specific "frozen version" of the versionable artifact. Version number is a system-generated monotonically increasing positive integer. Version label is a user assigned version string. The description is also assigned by the user.

3.8.4 Artifact Content

[122]
```
Identifiable ::= Sizable
Sizable =>
        Size                  : getSize():integer
```

Sizable defines the attribute Size common to artifacts that must provide the size information to enforce the container quota. Documents and messages are two classes of artifacts that support the Size attribute.

[122.1]
```
getSize : () -> integer
```

This function computes the size of the artifact.

[123]
```
Sizable ::= AttachmentHolder
AttachmentHolder =>
        Attachments           : {part SimpleContent}
```

Attachment holder defines the Attachments attribute common to artifacts that can hold the attached contents. Attachment holder includes discussion message, invitation, task, etc.

[124]
```
Sizable ::= Attachable
Attachable =>
        [Name]                : string
```

Attachable defines an aspect of Message and Document objects that allows them to be attached to the attachment holders.

[125]
```
MimeConvertable =>
        Size                  : getSize():integer,
        [ContentID]           : string,
        MediaType             : part MimeConvertableType
```

MimeConvertable defines an aspect of Message and Content objects that allows them to be converted to MIME format. The Size attribute is the size to be counted against the quota. The transfer length of the object may be different from the size used for quota allocation.

[126]
```
MimeConvertableType = MessageArtifact | MultiContentType
        | MIME-Content-Type | MIME-Multipart-Type
```

The enumeration of MIME convertable type. The constant MessageArtifact represents the Message objects that are MimeConvertable.

[127]
```
Identifiable ::= Content
MimeConvertable ::= Content
Content =>
        [MediaType]           : part ContentType
```

A content contains the data for a document or message. The content, simple content, and multi-content form a composite design pattern.

[128]
```
ContentType = MIME-Content-Type | MultiContentType
```

The enumeration of content type.

[129]
```
Content ::= SimpleContent
SimpleContent =>
```

```
        Data                    : <octet>,
        MediaType               : part MIME-Content-Type,
        [ContentEncoding]       : string,
        [CharacterEncoding]     : string,
        [ContentLanguage]       : locale
```

A simple content holds a single piece of data. The media type is an official RFC2046 type. Content encoding specifies the RFC2616 content encoding applied to the content. Character encoding specifies the RFC2616 character set of the content (a missing value means that the content should be treated as binary or raw). Content language specifies the RFC2616 content language for the content (a missing value means non-natural language content).

[130]
```
Content ::= MultiContent
MultiContent =>
        Parts                   : <part MimeConvertable>,
        MediaType               : part MultiContentType
```

A multi-content instance represents the multi-parts of a message or document. It is a composite content that can contain a list of simple or composite contents.

[131] `MultiContentType = Alternative | Related | Parallel | Mixed`

The enumeration of multi-content type, representing `multipart/alternative, multipart/related, multipart/parallel,` and `multipart/mixed.`

[132]
```
Content ::= OnlineContent
OnlineContent =>
        OnlineAttachment        : Ref Artifact
```

An online content holds the online artifact attached to a message or invitation. The online artifact must be rendered as a URL when the message or invitation is delivered to external recipients.

3.9 Metadata Management

3.9.1 Template

[133]
```
Artifact ::= Template
Template =>
        [Description]          : richtext
```

A template defines the reusable structures that can be replicated in new entities of an entity class.

[134]
```
Template ::= BasicTemplate
BasicTemplate =>
        EntitySchema           : Ref EntitySchema,
        AttributeTemplates     : {part AttributeTemplate}
```

A basic template uses the basic attribute templates to define the simple reusable structures.

[135]
```
Identifiable ::= AttributeTemplate
AttributeTemplate =>
        Definition             : ref AttributeDefinition,
        Mandatory              : boolean,
        Final                  : boolean,
        Prompted               : boolean,
        ForceDefault           : boolean,
        AllowedValues          : <part CollabPropery>,
        [DefaultValue]         : part PropertyValue,
        [MinimumValue]         : part PropertyValue,
        [MaximumValue]         : part PropertyValue,
        [MinimumValueInclusive] : boolean,
        [MaximumValueInclusive] : boolean
```

An attribute template is a template for creating an attribute according to a specific attribute definition. The settings for Mandatory, Prompted, ForceDefault, and DefaultValue must not violate the following two constraints: “if Mandatory and not Prompted then DefaultValue is not null” and “if ForceDefault then Mandatory and DefaultValue is not null.” The following table summarizes the possible combination of settings and user interface behavior.

Mandatory	Prompted	ForceDefault	DefautValue	UI Behavior
False	False	False	Can be null	The user will not be prompted but may supply the value in a field in the options tab.
False	True	False	Can be null	The user will be prompted to supply a value or leave the field null.
True	False	False	Is not null	The system will use the default value, which must not be null; user may override the default value in the options tab.

Mandatory	Prompted	ForceDefault	DefautValue	UI Behavior
True	True	False	Can be null	The user must supply a value or accept the default value if the default value is not null.
True	True or False	True	Is not null	The system will use the default value only. The default value will be displayed to the user if prompt is true. If prompt is false, the user may view the default value in options tab.

[136]
```
Template ::= AdvancedTemplate
AdvancedTemplate =>
        TemplateId              : string,
        Author                  : string,
        AuthorCreationTime      : timestamp,
        [CopyrightInfo]         : string,
        [ContactInfo]           : string,
        Definition              : part Content,
        TransportableFormat     : getTransportableFormat():Content
```

An advanced template contains an XML document content, in the Definition attribute, that prescribes the replicable structures, including nested structure of sub-entities, associations, dependencies, etc. The template id represents the unique vendor-specific id. The template author or author creation time could be different from the Creator or CreatedOn attributes of the template entity if, for example, the template is created by a third-party vendor. The template contact info is the vendor which created this template.

[136.1]
```
getTransportableFormat : () -> part Content
```

This function computes the XML document suitable for import, export, and transport of the associated template.

3.9.2 Marker

[137]
```
Artifact ::= Marker
Marker =>
        [Description]           : richtext,
        Entities                : getEntities():{Entity}
```

A marker is used by end users to group (categorize) entities together by a criteria meaningful to the users. Markers can be flat or hierarchical. Hierarchical markers are primarily used for maintaining taxonomies. A marker is considered part of the metadata of the entity. The marker can also have properties associated with it. In some cases when the user applies a marker to an entity, the marker is private such that only the user who applies the marker can browse or locate the entity through the marker.

[137.1]
```
getEntities : viewer -> {Ref Entity}
```

This function returns all entities, marked by the same marker, that are accessible to the given viewer.

3.9.3 Label

[138]
```
EntitySchema ::= LabelSchema
LabelSchema => ;
```

A label schema contains the attribute definitions for different types of labels.

[139]
```
Marker ::= Label
Label =>
        [EntitySchema]          : Ref LabelSchema,
        LabelApplicationCount   : integer
```

A label represents a keyword that is directly attached to an entity for the purpose of classifying the entity based on the label. Labels are non-hierarchical. Labels can be pre-defined labels shipped out of the box. Such labels are visible to all users within the enterprise. Labels can be defined by a user and visible only to the user who created them. The label schema represents the type of the label. For example, the label schema Recommendation can include labels by the name “Favorite,” “Important,” etc. Another label schema System can include labels for “Work,” “Personal,” “ToDo,” “Alert,” “PeopleList,” etc. A label of the System label schema is a pre-defined label in an enterprise scope. The system labels cannot be deleted and the names cannot be changed.

[140]
```
Identifiable ::= LabelApplication
LabelApplication =>
        LabeledEntity           : Ref Entity,
        Label                   : Ref Label,
        Type                    : part LabelApplicationType
```

A label application is an instance of association between a label and a specific entity.

[141]
```
LabelApplicationType = Public | Private
```

The enumeration of label application type.

3.9.4 Category

[142]
```
EntitySchema ::= CategorySchema
CategorySchema => ;
```

A category schema represents a taxonomy.

[143]
```
Marker ::= Category
Category =>
        [EntitySchema]          : Ref CategorySchema,
        DefaultTemplate         : part CategoryApplicationTemplate,
        Template                : getEffectiveTemplate(),
        [SuperCategory]         : Ref Category,
        SubCategories           : {Part Category},
        Attributes              : {part ref AttributeDefinition},
        Abstract                : boolean
```

A category is used to classify an entity under a structured taxonomy. The metadata administrator primarily creates and makes them available to all users within the enterprise. Categories are hierarchical in nature. The names of the categories within a category hierarchy must be unique. A category holds the category level attributes shared by all entities classified by the category. It uses one category application object per entity to hold the instance level attributes. A category application is created each time a category is applied to an entity.

Many category application templates can be associated with one category. For this reason, we need a function that will determine the effective template for a given container. The default category application template is used if there is no category application template configured for the container. The default category application template is created when the category is created.

[143.1]
```
getEffectiveTemplate : Container -> ref CategoryApplicationTemplate
```

This function computes the template that is in effect for the category when applied within a given container.

[144]
```
Identifiable ::= CategoryApplicationTemplate
CategoryApplicationTemplate =>
        Category                : Ref Category,
        CopyOnVersion           : boolean,
        Hidden                  : boolean,
        Final                   : boolean,
        Required                : boolean
```

A category application template captures the forms and default settings for the category applications. A category application template holds the attribute templates to instantiate the attributes of the category applications. Many category application templates can be associated with the same category.

[145]
```
Identifiable ::= CategoryApplication
CategoryApplication =>
        CategorizedEntity       : Ref Entity,
        Category                : Ref Category,
        Attributes              : {part Attribute}
```

A category application contains the instance-level attributes for the classified entity.

3.9.5 Bond

[146] *EntitySchema* ::= *BondSchema*
BondSchema => ;

A bond schema contains the attribute definitions for different types of bonds.

[147] *Entity* ::= *Bond*
Bond =>

```
        [EntitySchema]          : Ref BondSchema,
        [Description]           : richtext,
        Type                    : string,
        Attributes              : {part Attribute},
        Properties              : {part CollabProperty},
        [Root]                  : Ref Bondable,
        BondedEntities          : {part BondEntityRelation},
        Unread                  : isUnread():boolean,
        New                     : isNew():boolean,
        Recent                  : isRecent():boolean
```

A bond relates two or more entities to each other. The bond schema represents the type of the bond, such as “Discuss This,” “Follow-up,” and “Related Material.” All bonds can be annotated with arbitrary data via the Properties attribute. Refer to Artifact for the definitions of the change management functions isNew(), isRecent(), and isUnread().

The Root attribute can be used to define a hierarchy where the entity attached as the root is assumed to be a relative root above the entities listed among bonded entities.

Issue 1 *Note that entities among bonded entities may need to be a list (preserving the relative ordering of entities in the list). Ordered list can support certain searching capabilities.*

[148] *BondEntityRelation* =>

```
        Entity                  : Ref Bondable,
        Attributes              : {part Attribute},
        Properties              : {part CollabProperty}
```

A bond-entity relation allows a set of properties to be optionally associated with a particular entity in a bond relationship. This allows properties to be assigned to each edge(bond)-node(entity) combination in a bond.

3.10 Preference

[149]
```
Entity ::= PreferenceProfile
PreferenceProfile =>
        Name                    : string,
        [Description]           : richtext,
        PreferenceSets          : {PartRef PreferenceSet}
```

A preference profile contains a collection of preference sets that can be in effect for different circumstances. The user can activate one preference profile from an available list of preference profiles. For example, a user can switch back and forth between the regular operation profile and the business travel profile.

[150]
```
Entity ::= PreferenceSet
PreferenceSet =>
        Name                    : string,
        [Description]           : richtext,
        [Template]              : immutable Ref BasicTemplate,
        [ExtendsFrom]           : Ref PreferenceSet,
        Preferences             : {part PreferenceProperty}
```

A preference set is a container of various preference settings declaring the choices, desires, course of actions, or customizations. A preference set aggregates the related settings for some functional area. The preference sets can be configured for accessors (users and groups) and containers (enterprises, organizations, workspaces, and folders). Two preference sets are said to be compatible if they are based on the same preference set schema. One preference set can inherit preference properties from a compatible preference set according to the predefined inheritance rule. The ExtendsFrom attribute can be used to explicitly specify the preference set to inherit. Template provides the attribute templates that prescribe how to create the preference properties.

[151]
```
PreferenceProperty =>
        Name                    : immutable string,
        Value                   : PropertyValue,
        Type                    : getPropertyType(),
        [Format]                : string,
        Final                   : boolean
```

A preference property stores a preference setting for a preference set that can override the setting on another preference set. When a preference set inherits from another preference set, the latter is said to be at a higher scope. If the Final attribute is true, this preference setting cannot be overriden by any setting from the preference sets in the lower scopes.

[151.1]
```
getPropertyType : () -> PropertyType
```

This function computes the type of the property value in the associated preference property.

3.11 Workspace

[152]
```
Scope ::= Workspace
ArtifactContainer ::= Workspace
Lockable ::= Workspace
Workspace =>
        [Description]           : richtext,
        Elements                : {Part Folder},
        Parent                  : getParent():Community,
        [Trash]                 : Part Trash,
        [Inbox]                 : getInbox():Folder,
        [DefaultCalendar]       : getDefaultCalendar():Calendar,
        [DefaultTaskList]       : getDefaultTaskList():TaskList,
        [DefaultAddressBook]    : getDefaultAddressBook():AddressBook,
        [DefaultConference]     : getDefaultConference():Conference,
        QuotaStatus             : part QuotaStatus,
        [QuotaConfiguration]    : part WorkspaceQuotaConfiguration,
        PrimaryContact          : Ref Actor
```

A workspace is a scope that defines a logical scope of work for users. All workspaces must have a name (the Name attribute from Entity is not optional). Each workspace is addressable individually (and they may have a variety of ways of being addressed). Every workspace is created from a template that specifies an initial configuration for the workspace. Changes to the template can propagate to existing workspaces created from that template.

NOTE A workspace is a durable context and place to collaborate. One such context is the durable, asynchronous meeting place that draws several analogies to real-time meetings such as web/audio conferences. Like in a real-time meeting, a participant can observe the presence of other participants in a workspace meeting. Like in a real-time meeting, there are conversations in a workspace meeting. You can address email, voice, fax, IM (unified messages) to the workspace. Through asynchronous channels, workflows, subscriptions, etc., in the workspace, a workspace meeting adds more dimensions to the conversations. For example, real-time conversations can take place through web conferences, synchronous conversations can take place through chat rooms, and real-time and synchronously conversations are accessible asynchronously through conference or chat transcripts in the workspace. Asynchronous conversations can take place through discussion forums, blogs, messages, annoucements, calendar schedules, task assignments, documents, and wiki pages in the workspace. Workspace subscriptions, actionable alerts, and workflows add additional mode of conversations in the workspace.

[152.1] `getInbox : () -> Ref Folder`

This function determines the folder for incoming messages.

[152.2] `getDefaultCalendar : () -> Ref Calendar`

This function determines the default calendar for workspace.

[152.3] `getDefaultTaskList : () -> Ref TaskList`

This function determines the default task list for workspace.

[152.4] `getDefaultAddressBook : () -> Ref AddressBook`

This function determines the default address book for workspace.

[152.5] *getDefaultConference* : () -> **Ref** *Conference*

This function determines the default conference for workspace.

[153]
```
QuotaStatus =>
        ConsumedQuota           : integer,
        Status                  : getStatus():Status
```

A quota status provides the quota consumed and status.

[153.1] *getStatus* : () -> *Status*

This function computes the status of quota consumption.

[153.2] *Status* = *Normal* | *SoftQuotaReached* | *HardQuotaReached*

The enumeration of status.

[154]
```
AdvancedTemplate ::= WorkspaceTemplate
WorkspaceTemplate =>
        Domain                  : string
```

A workspace template is used to define a workspace. The domain defines the line of business such as transportation, health care, finance, etc.

[155]
```
Workspace ::= PersonalWorkspace
PersonalWorkspace =>
        [ReminderList]          : getReminderList():Folder,
        [SubscriptionList]      : getSubscriptionList():Folder,
        [NotificationList]      : getNotificationList():Folder
```

A personal workspace is a context owned by a user.

[155.1] *getReminderList* : () -> **Ref** *Folder*

This function determines the folder to hold personal reminders.

[155.2] *getSubscriptionList* : () -> **Ref** *Folder*

This function determines the folder to hold personal subscriptions.

[155.3] *getNotificationList* : () -> **Ref** *Folder*

This function determines the folder to hold notifications.

[156]
```
Workspace ::= TeamWorkspace
TeamWorkspace =>
        [DefaultAnnouncements]  : Ref Forum,
        [DefaultRole]           : Ref AssignedRole,
        ParticipationMode       : part ParticipationMode,
        ParticipantsGroup       : PartRef Group,
        Participants            : getParticipants():{WorkspaceParticipant}
```

A team workspace is shared amongst its participants. Participants are essentially those actors that have access to the workspace (via access control). The semantics around team workspaces are different from the semantics around personal workspaces. The participants can be assigned to “full member,” “coordinator,” or “viewer” roles.

[156.1] *getParticipants* : () -> {**part** *WorkspaceParticipant*}

This function computes the workspace participant objects that specify the assigned roles and participant settings.

[157] *ParticipationMode* = *Open* | *ApprovalRequired* | *InviteOnly*

The enumeration of participation mode of a team workspace, which can be open to all, by approval, or by invitation.

3.12 Time Management

3.12.1 Calendar

[158]
```
Folder ::= Calendar
Calendar =>
        Occurrences             : {Part Occurrence},
        Invitations             : {Part Invitation},
        TimeZone                : Ref TimeZone,
        InheritTimeZoneFromOwner: boolean,
        AllowDoubleBooking      : boolean,
        BookingBehavior         : part BookingBehavior,
        IncludeInFreeBusy       : boolean,
        FreeBusyModifiedOn      : timestamp,
        DefaultPriority         : part Priority,
        DefaultICalPriority     : integer,
        AvailableHours          : part ref AvailableHours,
        DeriveAvailableHoursFromOwnerWorkingHours: boolean,
        EnrollmentGroup         : Part DynamicGroup,
        EnrollmentType          : part EnrollmentType,
        CalDavResourceName      : string
```

A calendar is a container of time management artifacts such as occurrences and invitations. The elements of calendar are classified in two categories: owned entries and calendar contents.

The owned entries are instances of Occurrence that hold the actual state of a calendar component, but do not block off time in the calendar (i.e. are not considered in a normal day/week/month view or free-busy lookup). Owned entries are generally created by the owner of the calendar or someone working on his/her behalf. They are not intended to be manipulated directly by the participants.

The calendar contents are instances of Invitation. These are associated with occurrences. Invitations are automatically added to the participant's calendar when a new occurrence is created. Calendar contents are owned by the participant and initially located in their default calendar. The calendar contents hold the data that is owned by the participant (i.e. to set the participation status and transparency) and act as a proxy to access the occurrence data owned by the organizer.

The enrollment group is used to specify the enrollment list which allows all users in the group to be enrolled and allows any users to opt out.

[159]
```
BookingBehavior = Open | FirstComeFirstServe
```

The enumeration of booking behavior. Booking behavior is open (for double booking) or first come first serve.

[160]
```
EnrollmentType = Public | Private
```

The enumeration of enrollment type. Enrollment type is public or private. When the enrollment type attribute is public the invited enrollees are added as participants of the occurrence. When the enrollment type is private, the system will only add invitation artifacts to the participant calendars.

[161]
```
Identifiable ::= BaseOccurrence
AttachmentHolder ::= BaseOccurrence
BaseOccurrence =>
        [Name]                  : string,
        Participants            : {part OccurrenceParticipant},
        CompositeParticipants    : {part OccurrenceCompositeParticipant},
        [Description]            : richtext,
        [Organizer]              : part Participant,
        [InternalOrganizer]      : Ref Actor,
        [Status]                 : part OccurrenceStatus,
        Type                     : part OccurrenceType,
        [Priority]               : part Priority,
        [Location]               : part Location,
        [Url]                    : uri,
        [Equipment]              : <string>,
        Transparency             : part Transparency,
        iCalSequence             : integer,
        iCalUid                  : immutable string,
        [iCalPriority]           : integer,
        iCalClass                : string,
        iCalCategories           : <string>,
        hasPendingReminders      : hasPendingReminders():boolean,
        hasMultipleParticipants : hasMultipleParticipants():boolean,
        hasCompositeParticipants: hasCompositeParticipants():boolean,
        hasMultipleInstances     : hasMultipleInstances():boolean
```

BaseOccurrence defines the common attributes of the occurrence series, occurrence, invitation series, and invitation that represent some kind of event that occurs on a calendar. The internal organizer is an actor who organizes the occurrence on behalf of the external agent.

[161.1] `hasPendingReminders : () -> boolean`

This function computes whether there is at least one attached reminder with a reminder pending status.

[161.2] `hasMultipleParticipants : () -> boolean`

This function computes whether there is more than one participant.

[161.3] `hasCompositeParticipants : () -> boolean`

This function computes whether there is at least one composite participant.

[161.4] `hasMultipleInstances : () -> boolean`

This function computes whether the associated occurrence series object has more than one occurrence instance.

[162]
```
Entity ::= OccurrenceSeries
Bondable ::= OccurrenceSeries
BaseOccurrence ::= OccurrenceSeries
OccurrenceSeries =>
        Occurrences              : {Ref Occurrence},
        Recurrences              : part DateTimeRecurrenceSet,
        OriginalInclusionRule    : part DateTimeRecurrenceRule,
        Duration                 : timeoffset,
        ExplicitlyModifiedOn     : timestamp
```

An occurrence series is a set containing all the occurrences associated with the same event. Every occurrence, including a single instance event, is associated with an occurrence series.

The attributes of the occurrence series are used as default values when new occurrences are added or when expanding the recurrence rules at creation.

[163]
```
Artifact ::= Occurrence
BaseOccurrence ::= Occurrence
Occurrence =>
        Start                   : part DateTime,
        End                     : part DateTime,
        Series                  : immutable PartRef OccurrenceSeries,
        iCalRecurrenceId        : immutable part DateTime,
        ExplicitlyModifiedOn    : timestamp,
        [OccurrenceExceptionToSeries]: part ExceptionToSeries
```

An occurrence instance represents some kind of event that occurs on a calendar, such as a meeting, a day event, etc. An occurrence may also be part of a recurring event, in which case all the related occurrences are part of the same occurrence series.

Occurrences are meant to be manipulated only by the organizer (or someone that was granted special privileges). Occurrences are typically created in the organizer's calendar. When an occurrence is created the system will automatically create the invitation objects in the participants' default calendars (including the organizer if he/she is also an attendee). Invitations act as proxies to allow the participants to access the information in the source occurrence. Invitations are the objects to be rendered in a day/week/month view of the participant’s calendar or when querying the free or busy status of the participant.

The purpose of the occurrence object is to hold the information common to all the invitations. When an occurrence is modified, the system will automatically make the proper adjustments to the invitations.

[164]
```
BaseOccurrence ::= BaseInvitation
BaseInvitation =>
        Invitee                 : getInvitee():Addressable,
        InviteeProperties       : {part CollabProperty},
        InviteeTransparency     : part Transparency,
        InviteeParticipantStatus: part OccurrenceParticipantStatus,
        [InviteePriority]       : part Priority,
        [InviteeICalPriority]   : integer,
        [InviteeRepliedOn]      : timestamp,
        InviteeICalCategories   : <string>,
        InvitationSubmittedOn   : timestamp,
        EffectiveTransparency   : getEffectiveTransparency():Transparency,
        iCalClass               : string
```

BaseInvitation defines the common attributes of the invitation series and invitation.

[164.1] `getInvitee : () -> Ref Addressable`

This function computes the invitee of the invitation.

[164.2] `getEffectiveTransparency : () -> part Transparency`

This function computes the effective transparency of the invitation.

[165]
```
Entity ::= InvitationSeries
Bondable ::= InvitationSeries
BaseInvitation ::= InvitationSeries
```

```
InvitationSeries =>
        Source                 : immutable Ref OccurrenceSeries,
        Invitations            : {Ref Invitation},
        CalDavResourceName     : string,
        Duration               : timeoffset,
        [ExplicitlyModifiedOn] : timestamp,
        OriginalInclusionRule  : part DateTimeRecurrenceRule,
        Recurrences            : part DateTimeRecurrenceSet
```

An invitation series is a set containing all the invitations for the same participant that are associated with the same event. The invitation series is automatically managed by the system. Every invitation is associated with an invitation series.

Invitation series acts as a proxy of the source occurrence series. Through this proxy, the participant can set the participant status field to accept or decline all the invitations associated with a recurring meeting.

[166]
```
Artifact ::= Invitation
BaseInvitation ::= Invitation
Invitation =>
        Start                  : part DateTime,
        End                    : part DateTime,
        Source                 : immutable Ref Occurrence,
        Series                 : immutable PartRef InvitationSeries,
        iCalRecurrenceId       : immutable part DateTime,
        [ExplicitlyModifiedOn] : timestamp,
        [OccurrenceExceptionToSeries]: part ExceptionToSeries,
        [InvitationExceptionToSeries]: part ExceptionToSeries
```

An invitation is a place holder for an event that is placed into a calendar. An invitation is associated with a source occurrence. Invitations are automatically manipulated by the system according to the following business logic.

Invitations are created in the default calendar of every participant of a new occurrence. The participant status of new invitations is set to needs action. Invitations are automatically deleted if the associated occurrence is deleted or marked as deleted. Invitations are deleted if the associated participant is removed from the occurrence. Invitations are created if a participant is added to an existing occurrence. When an occurrence is rescheduled (i.e. when start, end or duration is modified), the participant status of the associated invitations are reset to needs action. The invitations will be restored if they are previously deleted.

Invitation acts as a proxy to the source occurrence and holds the participant's data such as the participant status, transparency, and reminders. Once the invitation is linked with a participant, any modification of the participant status will be reflected in the occurrence, by modifying the participant's reply field in the participant status of the occurrence.

[167] `OccurrenceType = Meeting | DayEvent | Holiday`

The enumeration of occurrence type.

[168] `ExceptionToSeries = IndirectlyModified | DirectlyModified`

The enumeration of exception to series. Exception to series indicates whether an occurrence or invitation is an exception, respectively, to the occurrence series or invitation series. If exception may be due to direct or indirect modifications to the occurrence or invitation.

3.12.2 Task

[169]
```
Folder ::= TaskList
TaskList =>
        Todos                    : {Part Todo},
        Assignments              : {Part Assignment},
        TimeZone                 : Ref TimeZone,
        DefaultPriority          : part Priority,
        InheritTimeZoneFromOwner: boolean,
        CalDavResourceName       : string
```

A task list is a container of task management artifacts such as todos and task assignments. The elements of task list are classified in two categories: owned entries and contents.

The owned entries are instances of Todo that hold the actual state of a task list component. Owned entries are generally created by the owner of the task list or someone working on his/her behalf. They are not intended to be manipulated directly by the participants.

The task list contents are instances of Assignment. These are associated with todos. Assignments are automatically added to the participant's task list when a new todo is created. Task list contents are owned by the participant and initially located in their default task list. The task list contents hold the data that is owned by the participant (i.e. participation status and transparency) and act as a proxy to access the data owned by the organizer.

[170]
```
Identifiable ::= BaseTodo
AttachmentHolder ::= BaseTodo
BaseTodo =>
        [Name]                   : string,
        Participants             : {part TodoParticipant},
        CompositeParticipants    : {part TodoCompositeParticipant},
        [Description]            : richtext,
        [Organizer]              : part Participant,
        [InternalOrganizer]      : Ref Actor,
        Status                   : part TodoStatus,
        [Priority]               : part Priority,
        [iCalPriority]           : integer,
        [Location]               : part Location,
        [Url]                    : uri,
        CompanyNames             : <string>,
        iCalUid                  : immutable string,
        iCalSequence             : integer,
        iCalClass                : string,
        iCalCategories           : <string>,
        Type                     : part TodoType,
        [Workflow]               : Ref Workflow
```

BaseTodo defines the common attributes of the todo series, todo, assignment series, and assignment that represent some kind of entry in a task list. The internal organizer is an actor who organizes the occurrence on behalf of the external agent.

[171]
```
TodoType = Task | WorkflowTask
```

The enumeration of the type of todo.

[172]
```
Artifact ::= Todo
BaseTodo ::= Todo
Todo =>
        [Start]                 : part DateTime,
        [Due]                   : part DateTime,
        ExplicitlyModifiedOn    : timestamp,
        [Completed]             : part DateTime,
        PercentComplete         : integer
```

A todo is a type of task list entry. It represents an instance of a task. A todo may be part of a recurring task represented by the todo series.

Todos are meant to be manipulated only by the organizer (or someone that is granted special privileges). Todos are typically created in the organizer's task list. When a todo is created the system will automatically create assignment objects in the participants' default task lists (including the organizer's if he/she is an assignee). Assignments act as proxies to allow the participants to customize the source todo. Assignments are rendered in the task view of a task list.

The purpose of the todo object is to hold the information common to all the assignments. When a todo is modified the system can automatically make the proper adjustments to the assignments.

[173]
```
BaseTodo ::= BaseAssignment
BaseAssignment =>
        Assignee                : getAssignee():Addressable,
        AssigneeProperties      : {part CollabProperty},
        AssigneeParticipantStatus: part TodoParticipantStatus,
        AssigneePriority        : part Priority,
        AssigneeICalPriority    : integer,
        AssignmentSubmittedOn   : timestamp,
        [AssigneeRepliedOn]     : timestamp,
        AssigneeBillingInfo     : string,
        AssigneeMileage         : string,
        AssigneeICalCategories  : <string>,
        iCalClass               : string
```

BaseAssignment defines the common attributes of assignment series and assignment.

[173.1] `getAssignee : () -> Ref Addressable`

This function computes the assignee of the assignment.

[174]
```
Artifact ::= Assignment
BaseAssignment ::= Assignment
Assignment =>
        Source                  : immutable Ref Todo,
        [Start]                 : part DateTime,
        [Due]                   : part DateTime,
        [AssigneeTimeAllocated] : timeoffset,
        [AssigneeTimeSpent]     : timeoffset,
        [AssigneeCompleted]     : part DateTime,
        [AssigneePercentComplete]: integer,
        [AssigneeStart]         : part DateTime,
        [AssigneeDue]           : part DateTime,
        [AssigneeSortOrdinal]   : integer,
        [AssigneeComment]       : string,
        [AssigneeWorkflowTaskId] : immutable string,
        [ExplicitlyModifiedOn]  : timestamp,
        CalDavResourceName      : string
```

An assignment is an element of a task list associated with the source todos. Assignments are automatically manipulated by the system according to the following business logic.

Assignments are created in the default task list of every participant of a todo. The participant status of new assignments is set to needs action status. Assignments are automatically deleted if the associated todo is deleted or marked as deleted. Assignments are deleted if the associated participant is removed from the todo. Assignments are created if a participant is added to an existing todo. An assignment is the proxy through which the participant can modify the data such as the participant status, time allocated, and time spent.

Once the assignment is linked with a participant, any modification of the participation status will be reflected in the todo (i.e., the participants reply to the organizer by modifying the participant status of the assignment).

3.12.3 Enumerations

[175]
```
ParticipantStatus = NeedsAction | Accepted | Declined | Delegated
Delegated =>
        DelegatedTo             : Ref Accessor
```

The enumeration of participant status.

[176]
```
OccurrenceStatus = Cancelled | Tentative | Confirmed
```

The enumeration of occurrence status.

[177]
```
OccurrenceParticipantStatus = ParticipantStatus | Tentative
```

The enumeration of occurrence participant status.

[178]
```
TodoStatus = NeedsAction | Cancelled | Completed | InProcess
```

The enumeration of todo status.

[179]
```
TodoParticipantStatus = ParticipantStatus | Completed | InProcess
        | WaitingOnOther | Tentative
```

The enumeration of todo participant status.

[180]
```
Transparency =Opaque | Transparent | Tentative | OutOfOffice
        | DefaultTransparency
```

The enumeration of transparency. The invitees can set the transparency of the invitations in their calendars.

3.12.4 Business Hours

[181] *AvailableHours* => ;

An available hours instance can be for business or personal..

[182]
```
AvailableHours ::= BusinessHours
BusinessHours => ;
```

A business hours instance can be customized based on the work shifts.

[183]
```
BusinessHours ::= WeekBusinessHours
WeekBusinessHours =>
        Shifts                 : <part WeekShift>
```

A week business hours instance contains a sequence of week shifts.

[184]
```
BusinessHours ::= MultiWeekBusinessHours
MultiWeekBusinessHours =>
        Start                  : part DateTime,
        AlternatingWeekBusinessHours: <part WeekBusinessHours>
```

A multi-week business hours instance is a composite of week shifts. The Start attribute indicates a point in time when the first element of availabilities is active. For subsequent weeks, the elements of alternating week business hours are selected in a round-robin fashion.

[185]
```
WeekShift =>
        Type                   : part ShiftType,
        StartDay               : part WeekDay,
        StartTime              : time,
        EndDay                 : part WeekDay,
        EndTime                : time
```

A week shift represents the days between the StartDay and EndDay attributes. The attributes StartTime and EndTime do not contain timezone information; the timezone is to be taken from the entity owning the business hours (i.e., actor or calendar).

[186] *ShiftType* = *RegularShift* | *ExtendedShift*

The enumeration of shift type. A regular shift represents the regular working hours (e.g. 9AM to 6PM). An extended hours represent times where the subject is typically not available, but can be if needed (e.g., 8AM to 9AM and 6PM to 8PM).

3.12.5 Timezone

[187]
```
Entity ::= TimeZone
Bondable ::= TimeZone
TimeZone =>
        Aliases                 : {part TimeZoneAlias},
        Rules                   : {part TimeZoneRule},
        [Coordinates]           : part Coordinates,
        [CountryCode]           : string,
        Common                  : boolean
```

A time zone defines the aliases used for this particular time zone in different namespaces (e.g., “America/Montreal” in the “TZ Database” name space), as well as all the rules that defines the transition between standard time and daylight saving time and vice versa. The CountryCode attribute specifies the upper-case, two-letter country code as defined by ISO-3166. The Name attribute of TimeZone entities will be set to the TZ database time zone identifier (e.g., America/Los_Angeles). A time zone must have at least one time zone rule. A time zone may have only one alias per namespace.

[188]
```
Entity ::= TimeZoneAliasMap
TimeZoneAliasMap =>
        Aliases                 : {part TimeZoneAlias},
        Direction               : part TimeZoneAliasMapDirection,
        Namespace               : string
```

A time zone alias map defines a mapping from the time management time zones to the aliases of a given namespace, or from the aliases of a given namespace to the time management time zones. A time zone alias map defines the mapping for a simple namespace.

[189]
```
TimeZoneAliasMapDirection = FromNamespace | ToNamespace
```

The enumeration of time zone alias map direction. A time zone alias map direction indicates whether the mapping is from the time management time zones to the aliases of a given namespace (ToNamespace), or from the aliases of a given namespace to the time management time zones (FromNamespace).

[190]
```
TimeZoneAlias =>
        Namespace               : string,
        Alias                   : string,
        TimeZone                : Ref TimeZone
```

A time zone alias provides an alternative identifier used for a given supported time zone (e.g., “America/Montreal” in the “TZ Database” namespace, “Eastern Standard Time” in the “Microsoft Windows” namespace).

[191]
```
TimeZoneRule =>
        [Name]                  : string,
        IsDaylightSavingTime    : boolean,
        StartYear               : part Year,
        [EndYear]               : part Year,
        UTCOffsetTo             : timeoffset,
        UTCOffsetFrom           : timeoffset,
        UTCOffsetRaw            : timeoffset,
        OnsetMonth              : part Month,
        OnsetDay                : part TimeZoneOnsetDay,
        OnsetLocalTime          : time
```

A time zone rule defines a transition between standard time and daylight saving time (or the other way around).

[192] `TimeZoneOnsetDay = TimeZoneOnsetFixedDay | TimeZoneOnsetMovableDay`

The enumeration of time zone onset day. The day on which a time zone rule takes effect could be specified in one of the following forms:

• the 5th of the month

• the last Sunday in the month

• the last Monday in the month

• first Sunday on or after the 8th

• last Sunday on or before the 25th

[193]
```
TimeZoneOnsetFixedDay =>
        Day                     : Day
```

A time zone onset fixed day is used to specify an onset day that occur on the same date every year (e.g., the 5th of the month).

[194]
```
TimeZoneOnsetMovableDay = TimeZoneOnsetOrdinalWeekDayOfMonth
        | TimeZoneOnsetRelativeWeekDayOfMonth
```

The enumeration of time zone onset movable day, which is used to specify an onset day varying in date from year to year.

[195]
```
TimeZoneOnsetOrdinalWeekDayOfMonth =>
        Position                : integer,
        WeekDay                 : part WeekDay
```

A time zone onset ordinal week day of month is used to specify an onset day that can be specified as the first (+1), second (+2) or last (-1) specific day of the week (e.g., Sunday) of the month.

[196]
```
TimeZoneOnsetRelativeWeekDayOfMonth =>
        Relation                : part DateRelation,
        Day                     : part Day,
        WeekDay                 : part WeekDay
```

A time zone onset relative week day of month is used to specify an onset day that can be specified as a day of the week that occur “before”, “on or before”, “after” or “on or after” a given day of the month (e.g., Sunday on or after the 8th).

[197] `DateRelation = Before | OnOrBefore | After | OnOrAfter | On`

The enumeration of date relation. Each type of date relation specifies a relation between two dates.

[198]
```
type Year is integer
type Month is integer
type Day is integer
```

The types Year, Month and Day are used to define years (e.g., 2006), months (1-12) and days of months (e.g., 1 to 31) in the Gregorian calendar.

[199]
```
WeekDay = Sunday | Monday | Tuesday | Wednesday | Thursday
        | Friday | Saturday
```

The enumeration of week day. A week day defines a specific day of the week in the Gregorian calendar (e.g., Monday).

3.13 Message

[200]
```
Artifact ::= Message
Sizable ::= Message
VirusScannable ::= Message
MimeConvertable ::= Message
Message =>
        Content                 : part MimeConvertable,
        [Sender]                : part UnifiedMessageParticipant,
        [DeliveredTime]         : timestamp,
        [Priority]              : part Priority
```

A message is a unit of conversation. It holds a simple content or multipart message contents in the Content attribute. It includes the sender and references. The DeliveredTime attribute holds the time when the message is delivered to a given recipient. The sent time of the message is represented by the user's created-on time of the artifact. The Name attribute holds the subject or title of the message.

3.13.1 Unified Message

[201]
```
Message ::= UnifiedMessage
Attachable ::= UnifiedMessage
UnifiedMessage =>
        MediaType               : part MediaType,
        Receivers               : {part UnifiedMessageParticipant},
        CCReceivers             : {part UnifiedMessageParticipant},
        BCCReceivers            : {part UnifiedMessageParticipant},
        ReplyTo                 : {part UnifiedMessageParticipant}
```

A unified message is a special type of message delivered electronically over a computer, voice, fax, and other networks.

[202]
```
MediaType = Email | Voice | Fax | Notification
```

The enumeration of media type. A unified message can be one of these types. Email is a type of message that is delivered electronically over a computer network. Voice is a type of message that contains a voice or audio stream. Fax is a type of message that contains an image transmitted via phone lines using the fax protocol. Notification is a type of message that delivers the notification of events.

3.13.2 Instant Message

[203]
```
Message ::= InstantMessage
InstantMessage =>
        Receivers               : {part Participant},
        ConversationId          : string,
        ClientSideId            : string,
        MediaType               : part InstantMessageType
```

An instant message is a special type of message for one-on-one, synchronous, usually text based, conversation. By default the media type is Chat. Other types such as FileTransfer convey special processing requirements.

[204] *InstantMessageType = System | Chat | Broadcast | Notification*

The enumeration of instant message media type.

[205] *System = FileTransfer | InfoQuery | Logout*

The enumeration of system instant message media type.

[206]
```
Identifiable ::= InstantMessageRecipient
InstantMessageRecipient =>
        InstantMessageReceptacle
        : getInstantMessageReceptacle():InstantMessageReceptacle
```

Instant message recipient defines the attribute to hold the instant message receptacle. If the recipient is offline, the instant messages are not delivered through this online receptacle, but instead are delivered to a designated offline folder.

[206.1] *getInstantMessageReceptacle* : () -> **Part** *InstantMessageReceptacle*

This function returns the instant message receptacle for the recipient.

[207]
```
Entity ::= InstantMessageReceptacle
InstantMessageReceptacle =>
        OnlineInstantMessageSets: {Part OnlineInstantMessageSet}
```

An instant message receptacle holds one online instant message set for each endpoint.

[208]
```
Entity ::= OnlineInstantMessageSet
OnlineInstantMessageSet =>
        Name                   : string,
        [Description]          : richtext,
        OnlineInstantMessages : getOnlineInstantMessages():<InstantMessage>
```

An online instant message set holds the instant messages that are in transit to the recipient's instant messaging client. These messages for online delivery need not be persistent. However, messages are persisted if they have been delivered and archiving is on, or if the receiver is offline.

[208.1] *getOnlineInstantMessages* : () -> <**Part** *InstantMessage*>

This function computes the instant messages in chronological order.

3.13.3 Discussion Forum

[209]
```
ArtifactContainer ::= DiscussionsContainer
DiscussionsContainer =>
        ViewCount              : integer,
        LastPost               : getLastPost():DiscussionsMessage
```

Discussions container defines the attributes common to forum and topic, both of which can contain discussions.

[209.1] *getLastPost* : () -> **Ref** *DiscussionsMessage*

This function computes the last posted message.

[210]
```
Folder ::= Forum
DiscussionsContainer ::= Forum
Lockable ::= Forum
Forum =>
        PopularTopics          : getPopularTopics():{Topic},
        ForumContents          : {Part DiscussionsContainer}
```

A forum is a collection of other forums and topics.

[210.1]
```
getPopularTopics : () -> {Ref Topic}
```

This function computes the popular topics by several criteria including the number of replies.

[211]
```
Folder ::= Topic
DiscussionsContainer ::= Topic
Lockable ::= Topic
Topic =>
        Messages               : {Part DiscussionsMessage}
```

A topic represents a conversation among forum members; it is structured as a collection of discussions messages. The discussions semantics may impose the topic messages to be sorted in chronological order or threaded by reply.

[212]
```
Topic ::= Announcement
Announcement =>
        ActivatesOn            : timestamp,
        ExpiresOn              : timestamp,
        AnnouncementStatus : getAnnouncementStatus():AnnouncementStatus
```

An announcement is a special topic for messages that are valid for a specified period of time, depending on the activated and expire on times.

[212.1]
```
getAnnouncementStatus : () -> part AnnouncementStatus
```

This function computes the status of the announcement.

[213]
```
AnnouncementStatus = Pending | Active | Expired
```

The enumeration of announcement status.

[214]
```
Message ::= DiscussionsMessage
AttachmentHolder ::= DiscussionsMessage
DiscussionsMessage =>
        [InReplyTo]            : Ref DiscussionsMessage
```

A discussion message adds discussion semantics to the message. The parent of the discussion message is the topic that contains the discussion message.

3.14 Conference

[215]
```
Template ::= ConferenceTemplate
ConferenceTemplate =>
        ConferenceSettings      : part ConferenceSettings
```

A conference template specifies a set of initial conference settings. It can hold a predefined set of groups, properties, and permissions. In most cases conference groups in a template do not have any participant assigned. Preconfigured conference groups can be used to create conferences with a fixed list of attendees.

The conference template can be created by users and stored in the user's workspace. Global templates can be made available through shared workspaces with some common conference configurations.

[216]
```
Folder ::= Conference
Conference =>
        [Template]              : Ref ConferenceTemplate,
        Status                  : part ConferenceStatus,
        [RunningSession]        : part ref ConferenceSession,
        EndedSessions           : <part ConferenceSession>,
        Settings                : part ConferenceSetting,
        LogEntries              : <ref ConferenceLogEntry>,
        Address                 : getConferenceAddress():Address
```

A conference specifies the current status, conference settings, sessions, and logs. The elements of the conference are conference transcript documents containing visual and audio information collected during the sessions. A conference can have one or more sessions that specify the demarcation and sequencing of the transcripts in the conference, but only one of the sessions can be running at any moment. If a conference is created from a template, it has a reference to the corresponding conference template for informational purposes. Attendee's privileges in a conference are specified by the conference settings attribute.

A participant of a conference is a principal that has sufficient privileges to join the conference. Each participant is authenticated and provided with a unique address, which is computed using the conference address function. Environment provides, among other attributes, the type of conference client. The system will taylor the conference address for different client environments.

There are two ways to authenticate a participant. Internal principals are authenticated by standard credentials and identified as themselves. External persons will be authenticated by non-standard way (conference will use its own authentication mechanism) and be identified by a name and an optional conference key. Temporary "anonymous" principal and actor instances will be used to represent the external participants.

A conference can be in a number of distict status. The initial status is conference not started which means that there are no conference sessions. The conference is started when a conference participant, who has sufficient privileges to start the conference, requests the conference address for the first time. During this process the system creates a new conference session in the conference. The reference to this session is placed in the RunningSession attribute.

[216.1]
```
getConferenceAddress : (Environment, ConferenceParticipant) -> Address
```

This function computes the conference address for a conference participant suitable for the connecting environment.

[217]
```
ConferenceStatus = ConferenceNotStarted | ConferenceWaitingForHost
        | ConferenceRunning | ConferenceHibernated
        | ConferenceFailed | ConferenceEnded
```

The enumeration of conference status.

[218]
```
Identifiable ::= ConferenceSession
ConferenceSession =>
        Conference                : Ref Conference,
        [EndStatus]               : part ConferenceSessionEndStatus,
        Recordings                : <Ref Document>,
        LogEntries                : getLogEntries():<ConferenceLogEntry>,
        StartTime                 : timestamp,
        [EndTime]                 : timestamp
```

A conference session specifies the demarcation and sequencing of the transcripts in the conference. A conference session can end with a different result which is captured in the session ending status. The conference transcript is made available after the end of a conference session. A transcript can contain visual and audio information collected during the session. The transcript document is a media file, identified by the media type of the document, that can be replayed by third party media players. The conference assigns an ACL to the transcript document based on the conference settings and conference ACL.

[218.1]
```
getLogEntries : ConferenceParticipant -> <ref ConferenceLogEntry>
```

This function computes the conference log entries from the conference session for each principal participating in the session.

[219]
```
ConferenceSessionEndStatus = SessionEndHostLeft | SessionEndHostAborted
        | SessionEndNoHost | SessionEndSystemError | SessionEndTimedOut
        | SessionEndByAdmin
SessionEndSystemError =>
        ErrorCode                 : string,
        Description               : string
SessionEndByAdmin =>
        Reason                    : string
```

The enumeration of conference session end status. It describes the various reasons why a conference session ended.

[220]
```
ConferenceLogEntry =>
        Conference                : Ref Conference,
        Session                   : ref ConferenceSession,
        Participant               : part ConferenceParticipant,
        EntryTime                 : timestamp,
        Type                      : part PropertyType,
        Value                     : part ConferenceVariant
```

A conference log entry contains a conference variant value as an entry. The security policy on the conference log can be set through the conference settings. The default policy allows the host to see all entries while an attendee can see only entries related to him or her. There are several conference log entry types.

[221]
```
ConferenceSettings =>
        ConferenceRoles         : {part ConferenceRole},
        Properties              : {part ConferenceProperty}
```

A conference settings instance includes a conference configuration (properties) and user rights. Generally, all settings are customized from the conference template during the conference instantiation phase. Conference settings are comprised of two sections. The conference roles section contains a list of roles created only for the lifetime of the conference instance. The roles are used to assign permissions to participants.

[222]
```
ConferenceRole =>
        Name                    : string,
        Participants            : {part ConferenceParticipant},
        ConferenceKeys          : {string},
        Properties              : {part ConferenceProperty}
```

A conference role is a named set of participants, which may be principals, actors, or groups, that have the same properties and permissions in the conference. Each conference role provides a match between a number of participants and a set of properties and permissions. Each participant may belong to a number of conference roles. In case of conflicts between the properties or permissions among two or more conference roles assigned to the participant, some well defined resolution rules will be applied. In case the participant is an external actor and cannot be authorized by the standard procedures, the conference key is used to identify the conference role.

[223]
```
CollabProperty ::= ConferenceProperty
ConferenceProperty =>
        Name                    : string,
        Value                   : part ConferenceVariant
```

A conference property is a special type of collab property that can hold special values, such as permissions and participants.

[224]
```
ConferenceVariant = PropertyValue | Grant | Deny
        | ConferenceParticipant | <ConferenceVariant>
```

The enumeration of conference variant. A conference variant includes standard property values extended by permission (Grant or Deny), conference participant, and array of conference variants.

3.15 User Subscription and Reminder

3.15.1 Subscription

[225]
```
EntitySchema ::= SubscriptionSchema
SubscriptionSchema =>
        Attributes              : {part ref AttributeDefinition},
        EntitySubscriptionRuleDefinitions
                                : {part SubscriptionRuleDefinition},
        ContainerSubscriptionRuleDefinitions
                                : {part SubscriptionRuleDefinition},
        SourceEntityClass       : entityClass
```

A subscription schema defines the attributes that can be part of the user subscriptions. There is at most one subscription schema for each source entity class, which can be a workspace, calendar, task list, forum, message, folder, etc. A subscription for entities of an entity class can be composed according to the entity or container level subscription rule definitions, depending on whether the subscription is attached to the container of the entity or to the entity itself. The container subscription rule definitions are used to compose a blanket subscription on all entities of an entity class in the container.

[226]
```
BasicTemplate ::= SubscriptionTemplate
SubscriptionTemplate =>
        EntitySchema            : Ref SubscriptionSchema,
        SubscriptionRules       : {part SubscriptionRule}
```

A subscription template defines a variant of the subscription schema by supplying the default settings or prescribing how to collect the settings for the required attributes. The subscription rules specify the actions, depending on the conditions around the events, source entities, and action doers, for automatic reaction or notification.

[227]
```
Artifact ::= Subscription
Subscription =>
        [Template]              : Ref SubscriptionTemplate,
        Attributes              : {part Attribute},
        Enabled                 : boolean,
        [Overrides]             : Ref Subscription,
        [AttachedTo]            : Ref Entity,
        Subscriber              : getSubscriber():User
```

A subscription contains the attributes submitted by a user, prescribing how to automatically react or notify the user when some events occur on an entity. A subscription on messages, for example, prescribes how to filter the messages.

A subscription refers to the subscription template that defines the metadata and forms used to construct the subscriptions. The user may enable or disable the subscription at any time. A subscription can be attached to a container to be applied to all entities of the given entity class under the container. It can be overriden by the subscription attached to a specific entity. The Overrides attribute refers to a container level subscription which it overrides.

[227.1]
```
getSubscriber : () -> Ref User
```

This function computes the subscriber who submitted the subscription.

[228]
```
Identifiable ::= SubscriptionRuleDefinition
SubscriptionRuleDefinition =>
        Name                    : string,
        [Description]           : richtext,
        Conditions              : {part SubscriptionConditionDefinition},
        Actions                 : {part SubscriptionActionDefinition}
```

A subscription rule definition specifies the available conditions and actions to compose the subscription rules.

[229]
```
Identifiable ::= SubscriptionConditionDefinition
SubscriptionConditionDefinition =>
        Name                    : string,
        [Description]           : richtext,
        Attributes              : {ref AttributeDefinition}
```

A subscription condition definition specifies the forms for a condition.

[230]
```
Identifiable ::= SubscriptionActionDefinition
SubscriptionActionDefinition =>
        Name                    : string,
        [Description]           : richtext,
        Attributes              : {ref AttributeDefinition}
```

A subscription action definition specifies the forms for an action.

[231]
```
SubscriptionRule =>
        Name                    : string,
        [Description]           : richtext,
        Definition              : ref SubscriptionRuleDefinition,
        Conjunctive             : boolean,
        Conditions              : {ref SubscriptionConditionDefinition},
        Actions                 : <ref SubscriptionActionDefinition>
```

A subscription rule specifies the conditions that must be satisfied and actions to be activated. The Conjunctive attribute specifies whether the conditions will be evaluated in conjunction. If conjunctive evaluation is used, all conditions must be satisfied to activate the actions. If the disjunctive evaluation is used, each condition is evaluated independently; the actions will be activated when any of the conditions are satisfied.

3.15.2 Reminder

[232]
```
Artifact ::= DefaultReminder
DefaultReminder =>
        [Container]             : Ref Container,
        [SourceEntityClass]     : entityClass,
        Trigger                 : part RelativeTrigger,
        Rule                    : part ReminderRule,
        Primary                 : boolean
```

A default reminder is defined on a container to be applied to all entities of the given entity class under the container.

[233]
```
Artifact ::= Reminder
Reminder =>
        [Source]                : Ref Entity,
        [DerivedFrom]           : Ref DefaultReminder,
        Trigger                 : {part ref TimedTrigger},
        [NextAbsoluteTriggerTime]: timestamp,
        Status                  : part ReminderStatus,
        Rule                    : part ReminderRule,
        Enabled                 : boolean,
        Primary                 : boolean
```

A reminder is an entity that is used to trigger a reminder action at some timed event. It associates a timed trigger on the entity. Reminders are adjusted when the default reminder changes because the entity is moved to another container, for example, when an invitation is moved from one calendar to another.

[234]
```
ReminderRule => ;
```

A reminder rule is a rule which must conform to a specifc rule definition associated with all reminders and default reminders.

[235]
```
ReminderStatus = ReminderPending | ReminderProcessed
```

The enumeration of reminder status. A reminder can be in pending or processed status.

[236]
```
TimedTrigger => ;
```

A timed trigger is a time-based trigger or stimulus that can set off an event. A timed trigger can be defined relative to the time of operations such as create, update, checkin, checkout, etc., or to a scheduled event (usually a future event) such as the start time of an calendar occurrence, due date of a task, end time of a conference, etc. A timed trigger can also be defined by absolute time independent of any operation.

[237]
```
TimedTrigger ::= RelativeTrigger
RelativeTrigger =>
        Offset                  : timeoffset
```

The relative trigger offset can be from any trigger, such as the start time, end time, anniversary, created on time, birth day, etc.

[238]
```
TimedTrigger ::= AbsoluteTrigger
AbsoluteTrigger =>
        Value                  : timestamp
```

The absolute trigger is a schedule for an alarm.

3.16 Workflow

[239]
```
EntitySchema ::= WorkflowSchema
WorkflowSchema =>
        Enabled                : boolean,
        Blocking               : isBlocking():boolean,
        Instances              : {Part Workflow},
```

A workflow schema defines a plan of actions. The schema provides the metadata about the attributes to customize the actions. In a typical deployment, a workflow schema corresponds to a workflow process (a unit of deployment in a workflow service engine) which can execute the workflows derived from the same workflow schema.

The workflow administrator creates and makes the workflow schemas available to all users within the enterprise. The workflow schema contains all instances, each of which holds the instance attributes and status.

The attribute definitions define the input and output parameters of the workflows. The Enabled attribute indicates whether the workflow process is available to execute the workflows derived from the associated workflow schema. The workflow may involve a mixed sequence of operations on the artifacts and interactions with some actors of the system. Hence the Blocking attribute indicates whether the workflow process involves any blocking operations, such as human interaction for approval.

[239.1]
```
isBlocking : () -> boolean
```

This function determines whether the workflows derived from the associated workflow schema would involve blocking operations. It may deduce this condition from the attribute definitions and other metadata.

[240]
```
BasicTemplate ::= WorkflowTemplate
WorkflowTemplate =>
        EntitySchema           : Ref WorkflowSchema,
        EventDefinition        : eventDefinition,
        Final                  : boolean,
        Mandatory              : boolean
```

A workflow template defines a variant of the workflow schema by supplying the default settings or prescribing how to collect the settings for the required attributes from the environment, including the actors, scopes, artifacts, etc. The EventDefinition attribute specifies the type of operations that initiate the workflow. Many workflow templates can be associated with the same workflow schema. The workflow configuration on the container configures the workflow templates and consequently binds the workflow schemas to the events in the container.

[241]
```
Artifact ::= Workflow
Workflow =>
        [Template]             : Ref WorkflowTemplate,
        [Description]          : richtext,
        Attributes             : {part Attribute},
        [Initiator]            : Ref Actor,
        Duration               : integer,
        Status                  : getWorkflowStatus():WorkflowStatus,
        [CurrentTodo]           : Ref Todo
```

A workflow is created from the workflow schema and template when the user initiates the workflow. The attributes contain the values that are used to create the workflow.

[241.1] *getWorkflowStatus* : () -> *WorkflowStatus*

This function computes the status of the workflow.

[241.2]
```
WorkflowStatus = Open | Closed | Failed
        | Aborted | Cancelled
```

The enumeration of workflow status.

4.0 References

[1] Berners-Lee, T., et al. “Uniform Resource Identifiers (URI): General Syntax”, RFC 2396, August 1998.

Beehive Object Model
January 2009

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.800.633.0973
Fax: +1.650.506.7200
oracle.com