The CoolAgent Ontology

A language for publishing and scheduling events

Craig Sayers and Reed Letsinger

HP Labs, Palo Alto

Abstract

This document describes a messaging language/ontology for scheduling events and handling calendar entries. It is intended for use in booking conference rooms, querying availability, scheduling meetings, and publishing upcoming events.

The design was influenced by iCalendar (RFC 2445), iTip (RFC 2446) and two early Internet drafts: draft-dawson-ical-xml-dtd-01 and draft-reddy-xml-ical-00.

It simplifies those prior implementations by considering a restricted domain (that of scheduling and publishing events). It advances those prior implementations by: allowing details of a proposed event to be included in a request for availability, allowing the expression of preferences for particular times/places, introducing a representation for uncertainty, and allowing for events which are both spatially and temporally distributed.

HP Labs Technical Report HPL-2001-194

August 2001.

Contents

1 Introduction

During the winter of 2001, the Agents for Mobility Department in the Software Technology Laboratory collaborated with members of Application Development Organization and Corporate IT to develop an agent-based meeting management system. The system, which we called *CoolAgent*, is organized as a distributed collection of software agents and services. The agents negotiate with each other to determine times and places best suited to the needs of both the meeting organizer and the meeting participants. For this negotiation to work, the software agents must be able to exchange messages containing information about meetings.

A simple description of a meeting would collect together answers to questions like: Where will the meeting be held? What time does it start? Who is invited? What is the purpose? This is the approach taken by conventional calendaring sytems. See for example: iCalendar [1], iTip [2] and two early Internet drafts: draft-dawson-ical-xml-dtd-01 [3] and draft-reddy-xml-ical-00 [4].

For our application, simply collecting a set of facts into a flat structure – perhaps as a set of attribute/value pairs – is not sufficient. This is best shown using a series of colloquial examples.

Representing the description:

> *"Local attendees should meet in the Eureka Conference Room, remote attendees should dial 1-800-123-4567 and enter code 890"*

requires handling events that are spatially distributed.

Representing:

> *"The presentation will be broadcast in California at 11:00am Los Angeles time, and in England at 11:00am London time".*

requires handling events that are temporally distributed.

Representing:

> *"I'd like to arrange a meeting with my team in Cupertino on Thursday morning, or in Palo Alto on Friday afternoon"*

requires handling the dependencies between times and places. These are clearly not independent. For example, an acceptable solution to that event request would not be a meeting in Cupertino on Friday afternoon.

Representing:

> *"I could meet at any time that day, but would prefer sometime between 10 and 11".*

requires handling the preferences and uncertainty which are a natural consequence of holding events with real people, in a real world.

To satisfy those requirements, we needed to represent the relationships between event times, places, and people; we needed to represent events that are physically or temporally distributed; and we needed to represent events during different stages of scheduling: starting from an initial imprecise description, and ending with an exact event suitable for publication.

In this remainder of this section, we'll introduce our representation of an event object, look at actions that operate on that object, and present an example conversation using those actions.

1.1 Event objects

Our solution was an ontology for events that is built on a single event object. Within each event object there are expressions for different instances of that event; where each instance is a particular combination of time/place/people. An example of a simple event object is shown in Figure 1.

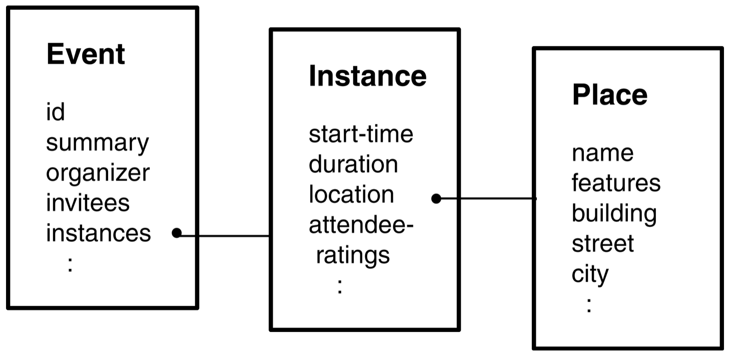

Figure 1 A simple example of an event. In this case, the event has a single instance at a single place.

That simple example contains three components:

Event - the highest level in the hierarchy, capturing details that are relevant for the whole event. It includes a unique id number, a summary, the name of the organizer, and a list of invitees. It also includes an instances expression. In this first, simple example, that expression is just a single instance.

Instance - describes a particular combination of time, place, and people. It includes expressions for the start-time, duration, location, and attendees. In this example, the expression for location refers to a single place.

Place - describes a particular place (for example, a conference room). It includes the address, phone number and a list of available features (for example, a whiteboard, or an overhead projector).

That example is simple, since the event contains only a single instance, and the location of that instance is only a single place. In practice, we need to represent events which are much more complex. To do that, we replace components with expressions.

For example, to represent an event that takes place in several times/places, we use an *all-of* relationship, so the single instance is replaced by an expression:

> *all-of* (instance1 instance2.... instanceN)

To represent the choices during the planning stages of an event, we have the additional expressions *one-of* and *any-of*. So again the single instance could be replaced by expressions:

> *one-of* (instance1 instance2.... instanceN)
>
> *any-of* (instance1 instance2.... instanceN)

And these expressions may be combined and nested to arbitrary levels. An example of an event using expressions is shown in Figure 2. This example is typical of an event during scheduling. It has been decided that the event will take place in both a single physical location and via a teleconferencing number. The exact physical location has been narrowed to two alternatives, but the final choice has not yet been made.

Notice that now the separation between event-specific, and instance-specific information becomes important. The higher-level event object captures information about the whole event, while the lower-level instance objects capture information specific to a particular instance.

1.2 Event Actions

The most common actions performed on an event object are: publish, inquire, schedule and cancel.

Publish - used to publish details of an event. This is used in cases where you don't expect to hear back from any invitees. For example, one might publish details of a public seminar.

Inquire - used to inquire about a potential event. This may be used during the early stages of scheduling an event, when you would like an indication of which times/places are likely to be well attended. It does not perform any scheduling, nor does it guarantee that a subsequent attempt to schedule will succeed. Think of it as a read-only operation. The result is an expression representing instances that were available for scheduling at the instant the inquiry was performed.

Schedule- used to actually schedule an event. The result is an expression representing the instances that were actually scheduled.

Cancel - used to cancel a previous action.

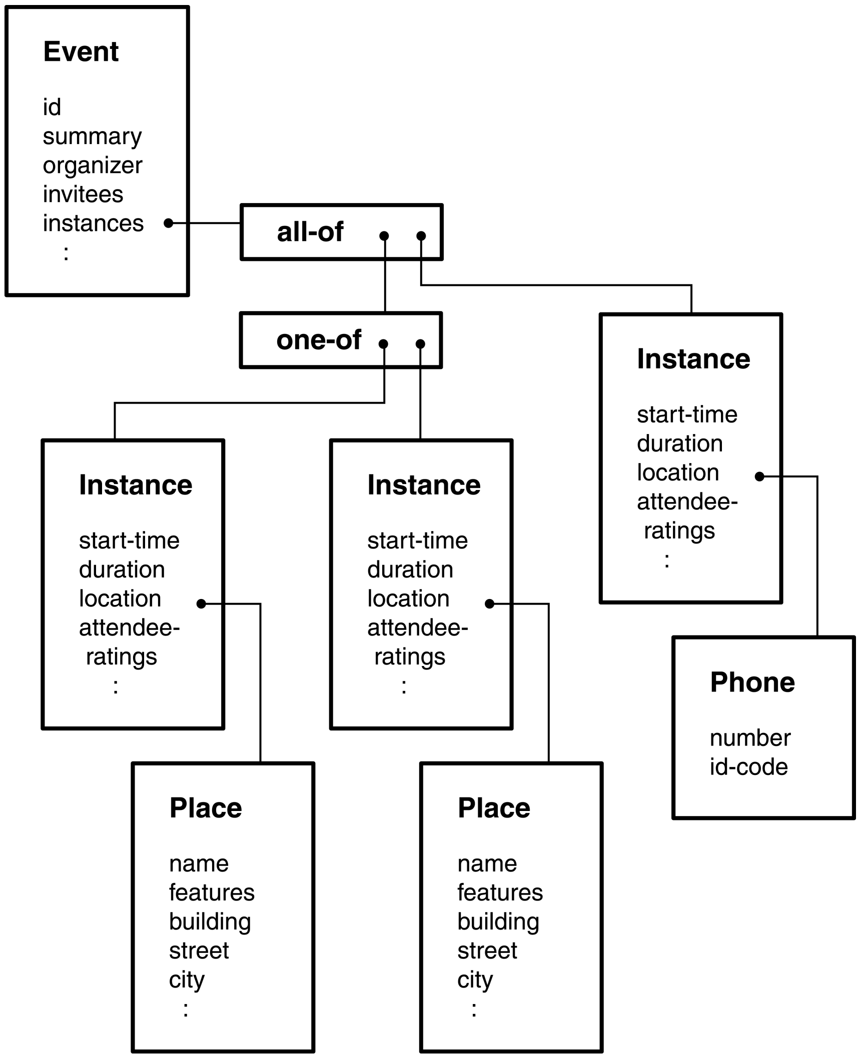

Figure 2 An example of an event described using the ontology. In this case, the event will take place in one of two physical locations and will also have a call-in number.

1.3 Example Conversation

One of the most common fragments of conversation is where an organizing agent inquires as to the availability of a set of invitee agents.

In this case, the organizer:

i. constructs an event object to describe the meeting, including an expression for potential instances (recall that each instance represents a particular combination of time and place)

ii. constructs a message to inquire about that event object

iii. sends that message to all invitees

Each invitee agent:

i. receives the request

ii. consults its owner's calendar and personal preferences to evaluate each potential instance.

iii. assigns a probability of attendance and a relative rating to each instance

iv. informs the organizer of the result.

The organizing agent then:

i. receives all the responses

ii. merges all of the result expressions to find an instances expression that optimizes the expected number of attendees and the expected rating. If there is no single suitable time/place, then it may decide to split the event (either spatially, or temporally) in order to meet the organizer's constraints.

iii. may choose whether to:

    a) try a different instances expression

    b) go ahead and actually schedule the meeting

    c) advise the organizer that no acceptable combination of time/place/people was possible

2 Messages

In our system, agents communicate by sending messages. Those messages are constructed using an Agent Communication Language [5], and the content of those messages is encoded using SL0 [6]. Those languages are both defined by the Foundation for Intelligent Physical Agents (FIPA) consortium.

2.1 The Agent Communication Language

The FIPA Agent Communication Language (ACL) is used to construct messages sent between agents. It commonly includes the sending and receiving agent names, a conversation identifier, the message content, and the names of the language and ontology used to encode that content. In our case, the message content contains descriptions of meetings encoded in SL0 using our ontology.

2.2 The SL0 language

In subsequent sections, we'll present detailed specifications using the language SL0. For the benefit of readers who may be unfamiliar with that language, the following is a brief overview.

In SL0, objects are described by terms. The most elementary terms are strings and numbers:

```
“Pat”

3.1415
```

Those terms may be collected in a set or sequence

```
(set “John” “Susan” “Pat”)

(sequence 2 4 6 “eight” )
```

(The distinction between sets and sequences occurs when messages encoded in this language are interpreted. The set does not imply any ordered relationship on its members, while the sequence does).

New collections of terms may be introduced using either unnamed or unnamed parameters:

```
(person “John” “Smith”)

(person
   :first “John”
   :surname “Smith”)
```

And they may be combined and nested to arbitrary levels:

```
(married
    :husband (person “John” “Smith”)
    :wife (person “Susan” “Jones”) )
```

Actions are described using the special term: “action”:

```
(action actor-term content-term)
```

The *actor-term* names the agent to perform the action, while *content-term* describes the action to be performed. SL0 does not specify the structure of content-terms, but it is natural to compose such terms out of an active verb followed by descriptions of objects the action applies to. For example, a possible content-term is:

```
(register
   (married
    :husband (person "John" "Smith")
    :wife (person "Susan" "Jones") ) )
```

and an action description with this content might be:

```
(action
  (agent-identifier
    :name registrar)
  (register
    (married
      :husband (person "John" "Smith")
      :wife (person "Susan" "Jones") ) ) )
```

SL0 provides two special predicate symbols, “done” and “result”:

```
(done (action action ) )

(result (action action ) result )
```

The former indicates that the action was performed successfully, while the latter indicates that the action is complete, and includes a result. For example, the expression:

```
(done
  (action
    (agent-identifier
      :name registrar)
    (register
      (married
        :husband (person "John" "Smith")
        :wife (person "Susan" "Jones") ) ) ) )
```

states that the registrar has registered the marriage.

SL0 allows arbitrarily complex descriptions of objects to be built up. However, it provides no support for combining simple facts into more complex facts using logical operators, such as ‘or’, ‘and’, ‘not’, ‘all’, ‘some’. If it is important to express these logical notions, either a richer content language needs to be used – FIPA defines languages SL1, SL2, and SL for this purpose – or the effects of logical operators need to be encoded using the mechanisms supported by SL0. In the CoolAgent application, the latter approach was adopted.

2.3 An XML encoding

In subsequent sections we'll employ the SL0 language. However, readers should appreciate that the exact same concepts would work equally well if the expressions were encoded in XML.

One approach to this is simply to use XML to encode the SL0 expressions. At this time, there is no official FIPA specification for an XML encoding of SL0, however, one simple (non-reversible) transform is to directly map each SL0 expression to an equivalent XML one. For example:

FIPA SL0		XML encoding of SL0
constant	→	*constant*
(set *terms* **)**	→	**<set>** *terms* **</set>**
(sequence *terms* **)**	→	**<sequence>** *terms* **</sequence>**
(name *terms* **)**	→	**<name>** *terms* **</name>**
:namedParam *term*	→	**<namedParam>** *term* **</namedParam>**

3 Object specifications

This section presents a specification for objects in the event ontology.

3.1 CoolEvent

This describes a single event. For example: a meeting.

```
(CoolEvent
   :id String
   :sequence Integer
   :organizer CoolAgent
   :contact CoolAgent
   :type String
   :summary String
   :invitees CoolAgentsExpr
   :instances CoolInstancesExpr
)
```

The *id* string is a unique identifier for this particular event. For example, a suitable *id* would be a 32 character (128-bit) GUID. All subsequent messages referring to the same event must include the same *id* token.

The *sequence* integer is incremented by one for each subsequent message sent by the meeting organizer (or their agent). This is used to match a reply to a request, as well as for distinguishing between revisions to the same meeting. Think of it as a version number for the meeting.

The combination of id/sequence can be encoded as a string, and used in the description/notes portion of a roombooker reservation, or an exchange calendar entry.

The *organizer* is the person organizing the event. For now, CoolAgent is synonymous with "Agent-Identifier".

The *contact* is the person who is responsible for arranging the meeting. It could be the organizer, or his or her delegate.

The *type* is the type of event – the only example right now is "meeting".

The *summary* is a short description of the event (suitable for a calendar entry).

In the future, we'll add many more event details (a URL, links to the presentations, transcripts….).

The *invitees* is a *CoolAgentsExpr*. This corresponds to one person, or group of people, invited to the event. Note that the organizer is not an invitee unless he/she is listed here.

The *instances* is a *CoolInstancesExpr*. This corresponds to one, or more, expected instances of the event. Each instance is a particular combination of time/place/people.

3.2 CoolAgent

For now, the CoolAgent is just a Jade Agent-Identifier.

3.3 CoolAgentsExpr

This is an expression describing one, or more agents.

```
CoolAgent |
one-of( set( CoolAgentExpr*) ) |
any-of( set( CoolAgentExpr*) ) |
all-of( set( CoolAgentExpr*) ) |
empty
```

3.4 Empty

This means "none".

Note that *empty* isn't the text "empty" it's literally an empty spot. For example:

```
one-of( set( ACoolAgentExpr )) is:
one-of( set() ) when ACoolAgentExpr == empty
```

3.5 CoolInstancesExpr

This may be a single instance, or an expression for a set of instances.

```
CoolInstance |
one-of( set( CoolInstancesExpr*) ) |
any-of( set( CoolInstancesExpr*) ) |
all-of( set( CoolInstancesExpr*) ) |
empty
```

3.6 CoolInstance

This is used to identify, and evaluate, a particular instance of an event – a particular combination of time and place (and, optionally, people).

```
( CoolInstance
   :start CoolDateExpr
   :duration CoolPeriodExpr
   :location CoolLocationExpr
   :attendee-ratings (set CoolAgentFloatPair*)
   :attendee-probabilities (set CoolAgentFloatPair*)
)
```

The *start* is the starting instant for this instance.

The *duration* is the length of the event.

The *location* is the location of this instance of the event.

The optional *attendee-ratings* is a set of ratings given to this particular event-instance by particular agents. An agent may indicate its owner's rating by including its own ID and rating value in this set.

The optional attendee-probabilities is a set of probability-of-attending given to this particular event-instance by particular agents. An agent may indicate its owner's likelihood-of-attending by including its own ID and expected probability value in this set.

3.7 CoolDateExpr

This represents an instant in time. It may be a fixed value (represented by a single value), or a range of values.

```
CoolDate |
range-of ( CoolDate CoolDate )
```

3.8 CoolDate

This represents an instant in time.

```
( CoolDate
   :string DateString )
```

In this implementation, the date must be specified as a string to avoid a Jade bug. The format for the string is FIPA SL0 Date in UTC (see later section on date-time specification for details)

3.9 CoolPeriodExpr

This represents a period of time. In the current implementation, the only legal duration is an Integer representing time in seconds, but future implementations will permit an expression for the period of the meeting (representing a range/selection of possible durations).

```
Integer
```

3.10 CoolLocationExpr

This represents the possible locations for an event instance. Each event instance will ultimately have just a single location, but several alternatives may be considered during the scheduling process.

```
CoolPhone |
CoolPlace |
one-of ( (set CoolPlaceExpr*) )
```

3.11 CoolPhone

This is a specification for a telephony connection.

```
( CoolPhone
   :number String
   :id String )
```

Where the telephone number is specified as a string, using the international format:

```
+countryCode-areaCode-localNumber
```

Where the *localNumber* may contain hyphens or spaces.

The *id* String is optional – it is indented to hold a teleconferencing code number. It should hold just the number, with no additional characters/comments.

3.12 CoolPlace

This is a physical place. For example, a conference room.

```
( CoolPlace
   :name String
   :capacity Integer
   :features CoolPlaceFeaturesExpr
   :area String
   :floor String
   :building String
   :site String
   :street String
   :locality String
   :region String
   :postal-code String
   :country String
   :latitude Float
   :longitude Float
   :telephone-number String
   :url String
)
```

The name is an optional string representing the name of this place. For example: “Enigma conference room” or “Pat’s cubicle” or “The barbeque area”.

The capacity is an optional integer representing the maximum number of people this place can support. (In practice, the actual capacity may be lower, depending on the seating style).

The features is an expression (see below).

The area, floor, and building, describe the location of this place within a particular site. The postal-code, region, locality, and street, describe the address of that site within the specified country. The country is a two-letter code, as specified in ISO 3166-1. For example:

The latitude and longitude are optional floating-point values representing the global position of this “place”.

The telephone number is an optional string in the international format:

```
+countryCode-areaCode-localNumber
```

Where the *localNumber* may contain hyphens or spaces.

The url is a web page associated with that place.

An example CoolPlace is:

```
(CoolPlace
  :name "Enigma"
  :capacity 7
  :features (all-of (set "whiteboard(2)" "overhead-projector"
                         "internal-ethernet"
                         "internal-wireless-802.11b"
                         "polycom"))
  :area "F10"
  :floor "U"
  :building "1"
  :site "HP Labs"
  :street "1501 Page Mill Rd"
  :locality "Palo Alto"
  :region "CA"
  :postal-code "94304-1126"
  :country "US"
  :telephone-number "+1-650-852-8674"
)
```

Note that all the extries in a CoolPlace are optional. It is quite acceptable to have an expression:

```
(CoolPlace)
```

indicating that you need to meet in a physical place, but the details of that place have yet to be determined.

3.13 CoolPlaceFeaturesExpr

This represents the features of a CoolPlace.

```
String |
one-of ( (set CoolPlaceFeaturesExpr*) )
any-of ( (set CoolPlaceFeaturesExpr*) )
all-of ( (set CoolPlaceFeaturesExpr*) )
```

Each possible string describes a particular feature.

Example features are:

```
conference-table
whiteboard
electronic-whiteboard
flipchart
overhead-projector
35mm-projector
video-projector
internal-ethernet
external-ethernet
internal-wireless-802.11b
external-wireless-802.11b
polycom.
```

Each feature string may optionally be terminated by an integer in parenthesis indicating the number of those features in the room.

Consumers of information described in the CoolPlace should, of course, recognize that the listed room details are only as accurate as the database from which they were obtained.

An example features expression is:

```
 (all-of (set "whiteboard(2)" "overhead-projector"
              "internal-ethernet" "internal-wireless-802.11b"
              "polycom"))
```

3.14 CoolAgentFloatPair

This is used to store the relationship between a particular agent and some floating-point value.

```
    ( CoolAgentFloatPair
           :agent CoolAgent
           :value Float )
```

For example, it is used to associate an agent with their probability of attending an instance of an event.

4 Action specifications

This section presents a specification for actions in our event ontology. Each action acts on objects (see previous section).

4.1 publish-event

Publish is used to notify others of an object's existence. For example, one might publish details of up-coming talks.

```
(publish-event Coolevent)
```

The id, sequence, contact, summary and at least one instance are required for this.

4.2 inquire-event

This is used to check on availability for an event during one, or more, time periods.

```
(inquire-event Coolevent)
```

The id, contact, summary and at least one instance are required for this.

4.3 schedule-event

This is used to schedule an event, or to update a previously-scheduled event.

```
(schedule-event Coolevent)
```

This is similar to inquire-event, it takes similar arguments, returns a similar response. The difference is that this actually attempts to make a booking. In addition to the parameters for inquire-event, this will usually require both an id and sequence number (the only exception being the very first request from the organizer to the meeting agent).

4.4 inquire-scheduled-events

This is used to find previously-scheduled events in a period between two instants in time.

```
(inquire-scheduled-events range-of( CoolDate CoolDate ) )
```

It returns a set of *CoolEvents*, where each CoolEvent is scheduled to occur at some point in the specified time period. Specifically, if the query period is range-of(A B), then the ending Time for each CoolEvent is guaranteed to be greater than A, and the starting time for each is guaranteed to be less than B. Note that the period of the event is not required to be entirely contained within the query period; it just needs to overlap it.

4.5 cancel-event

This is used to cancel a previous booking. It is not necessary to cancel a previous inquire-event action, but may be desirable (processing an inquiry can be an expensive operation so, if the results are no longer needed, its worthwhile to cancel it).

```
(cancel-event Coolevent)
```

4.6 translate-to-nl

This is used to translate an object to its natural-language representation.

```
(translate-to-nl (publish-event Coolevent) ) |
(translate-to-nl (result
                      (action actor (schedule-event Coolevent))
                      CoolInstancesExpr ) )
```

For now, the only legal objects that may be translated are a published event and the result of scheduling an event. In that case, the *actor* is the name of the agent asked to perform the scheduling operation, and the *CoolInstancesExpr* is the result of the scheduling action.

5 Date-time specifications

Unfortunately, both the XML schema specification, and the FIPA specification define slightly different date formats, and each allows more than one variation. So, to simplify things, we specify one representation native to each format.

In XML representations, we require use of the canonical form of timeInstant from the XMLSchema (which is a derived from ISO 8601). Note that date/time is always stored in UTC or "Zulu" time. It is never local time (unless you happen to be in England in the non-summer months!).

```
YYYY-MM-DDThh:mm:ssZ
```

In SL0 representations, we require the FIPA standard form, again in UTC:

```
YYYYMMDDThhmmssSSSZ
```

In either case:

YYYY is the year, generally represented by 4 digits using the Gregorian calendar.

MM is the month, January is 01, February 02…

DD is the day of the month, 01, 02, …

hh is the hour: 00, 01, …23

mm is the minute: 00, 01, … 59

ss is the seconds: 00, 01, … 59 (or 60 in the rare case of a "leap second").

SSS are the miliseconds

Examples in XML form:

Noon on New Year's Day, 2000 in England is:

`2000-01-01T12:00:00Z`

Noon on New Year's Day, 2000 in New York is:

`2000-01-01T07:00:00Z`

Note that we never send time as local time. This is deliberate. Local time depends on the machine on which you happen to be running, and that may not match either the place where the event is expected to occur, or the place from which each invitee will attend.

6 Messaging patterns

In this section, we present some typical examples of messaging patterns using the ontology. These are described from the point-of-view of three different agents: a personal agent (which represents a person), a meeting agent (which represents a meeting) and a room broker agent (which represents a set of physical rooms).

6.1 Personal Agent

The personal agent represents a human within the agent world. In the case of scheduling events using this ontology, the agent serves two distinct roles. As the representative of a human who is organizing an event, it initiates a conversation with the meeting agent factory. As the representative of a human who has been invited to an event, it responds to inquiries and scheduling requests.

6.1.1 Generating a schedule-event for the Meeting Agent Factory

Use this to ask the meeting agent factory to create a new meeting agent to schedule an event. Send a message:

```
REQUEST (action (schedule-event coolEvent))
```

And expect the newly-created meeting agent to respond with a series of:

```
INFORM (result (action (schedule-event coolEvent)) coolEvent)
```

The very first response comes back almost immediately and includes the event id, subsequent responses contain updated information (as the meeting agent hears back from invitees).

6.1.2 Generating a cancel-event for the Meeting Agent

Use this to ask the meeting agent to cancel a previously-scheduled meeting. Send a message:

```
REQUEST (action (cancel-event coolEvent))
```

And expect the response:

```
INFORM (done (action (cancel-event coolEvent)))
```

If, instead, the response is:

```
REJECT (action (cancel-event coolEvent))
```

then the meeting agent did not recognize the event (its eventId did not match).

6.1.3 Processing an inquire-event from the Meeting Agent

The meeting agent uses this to test if an event is practical. The personal agent should not actually schedule the event, but must check on availability. There is no guarantee that an inquiry will be followed by any subsequent message from the meeting agent.

```
REQUEST (action (inquire-event coolEvent))
```

For each entry in the eventInstanceExpr, the personal agent should attempt to satisfy that expression, looking up the date-time from that eventInstance, and inserting a probability-of-attending and optional rating (inserting its own attendee information into each).

If the expression could be satisfied, it should return:

```
INFORM (result (action (inquire-event coolEvent))
eventInstancesExpr)
```

`Otherwise, it should return:`

```
REJECT (action (inquire-event coolEvent))
```

6.1.4 Processing a schedule-event from the Meeting Agent

This is used when the meeting agent wishes to actually schedule an event, or to update a previously-scheduled event. The personal agent should expect:

```
REQUEST (action (schedule-event coolEvent))
```

The personal agent should first check if the enclosed event id exists in their calendar.

Assuming it does not, it should attempt to satisfy the instancesExpr by booking each required date/time/duration, storing the eventId in each calendar entry.

If the bookings succeeds in satisfying the expression, then it should insert attendee info into that eventInstance, and return:

```
INFORM (result(action (schedule-event coolEvent))
eventInstancesExpr)
```

If the expression is not completely satisfied, then it should remove any related calendar entries, and return:

```
REJECT (action (schedule-event coolEvent))
```

6.1.5 Processing a cancel-event from the MeetingAgent

This is used to cancel a previously scheduled booking. Expect to receive:

```
REQUEST (action (cancel-event coolEvent))
```

The personal agent should look for the enclosed event id in their calendar, and delete any matching entries. It should then reply either:

```
INFORM (done (action (cancel-event coolEvent)))
```

Or, if there was no matching entry:

```
REJECT (action (cancel-event coolEvent))
```

6.2 Meeting Agent

The meeting agent represents a particular event. It is created when the organizer of the event first requests a new meeting, and remains in existence for at least the lifetime of the meeting (and possibly longer, if it were also to store notes or details pertaining to the meeting).

6.2.1 Processing a schedule-event from the Personal Agent

This is used to actually schedule an event, or to update a previously-scheduled event. Expect to receive:

```
REQUEST (action (schedule-event coolEvent))
```

(That message will initially be sent from the personal agent to the meeting agent factory, which will create a new meeting agent, and then forward it the request).

The meeting agent should respond immediately with:

```
INFORM (result (action (schedule-event coolEvent)) coolEvent)
```

adding a generated event and sequence id.

It should be expected that a number of subsequent INFORM results will be generated by the meeting agent, each one updates/replaces the preceeding one, adding details to the eventInstances. This will occur as the personal agents for each invited attendee send in their responses and/or change their responses.

If, at any time, it becomes clear the meeting can no longer be scheduled, then it should return:

```
REJECT (action (schedule-event coolEvent))
```

6.2.2 Processing a cancel-event from the Personal Agent

This is used to cancel a previously scheduled booking. The meeting agent should expect to receive:

```
REQUEST (action (cancel-event coolEvent))
```

It should then verify the enclosed eventId matches the eventId for the meeting it owns. If it does not match, it should return:

```
REJECT (action (cancel-event coolEvent))
```

Otherwise, it should forward that message to each agent it contacted about this event.

And expect to receive a number of responses that are of the form:

```
INFORM (done (action (cancel-event coolEvent)))|
REJECT (action (cancel-event coolEvent))
```

It should then reply back to the personal agent:

```
INFORM (done (action (cancel-event coolEvent)))
```

6.3 Room Broker Agent

The room broker serves to represent physical rooms within the agent world. When one, or more, rooms are needed for an event, the meeting agent contacts the room broker to check on availability and schedule rooms.

6.3.1 Processing an inquire-event from the Meeting Agent

This is used while the meeting agent is testing to see if an event is practical. The room broker shouldn't actually schedule anything, but must check on availability. There is no guarantee that an inquiry will be followed by any subsequent message. The room broker should expect to receive:

```
REQUEST (action (inquire-event coolEvent))
```

It should then attempt to satisfy the instances expression, finding suitable rooms as necessary. Each room must satisfy all the constraints requested (start >= start, end <= end, duration == duration, roomFeatures >= roomFeatures).

If the entire expression could be satisfied, then it should return:

```
INFORM (result (action (inquire-event coolEvent))
eventInstancesExpr)
```

Otherwise, it should return:

```
REFUSE (action (inquire-event coolEvent))
```

6.3.2 Processing a schedule-event from the Meeting Agent

This is used to actually schedule an event, or to update a previously-scheduled event. The room broker should expect to receive:

```
REQUEST (action (schedule-event coolEvent))
```

Given an eventInstancesExpr, it should follow the same procedure as above for finding available rooms/times, except actually attempt to make a booking for each room. Within each booking it should record the eventId (we'll need that later, if its necessary to cancel the booking). If the entire expression was satisfied, then send

```
INFORM (result(action (schedule-event coolEvent))
eventInstancesExpr)
```

Otherwise, cancel any bookings related to this event, and return

```
REJECT (action (schedule-event coolEvent))
```

If the eventId already exists in the database, then this is a change to an existing meeting.

In that case, it should appear as though we are doing the following:

i) lock database

ii) free existing booking

iii) attempt to make new booking using that list.

iv) if unsuccessful, then reinstate original booking

v) unlock database

vi) return result.

6.3.3 Processing a cancel-event from the Meeting Agent

This is used to cancel a previously-scheduled booking.

```
REQUEST (action (cancel-event coolEvent))
```

The room broker agent should look for eventId in the database and delete any matching entries. It should then reply:

```
INFORM (done (action (cancel-event coolEvent)))
```

Or, if there was no matching entry, it should return:

```
REJECT (action (cancel-event coolEvent))
```

7 Processing expressions

This section discusses our implementation of the expression processing.

Recall from the introduction that our ontology allows a single event instance to be replaced by an expression of arbitrary complexity. To simplify the processing of such expressions, we introduce a classic mapping function of type:

```
map :: (function, expression) → expression | null
```

Where *function* is of type:

```
function :: instance -> instance | null
```

This takes a function that acts on a single instance, and maps it over an *expression*. If the expression could be satisfied, then the resulting *expression* is returned, otherwise the result is null.

The idea is that the writer of a service need only write a function to handle processing a single instance. Then they can use the mapping function to process any instances expression.

The definition of map is shown in Figure 3.

This implementation uses a depth-first search to move through the expression. This works well for our implementation, but note that it may fail to find a solution, even if one exists. For example:

```
map( schedule, all-of( one-of(roomA,roomB), roomA) )
```

This has a solution:

```
all-of(roomB, roomA)
```

But the mapping function will fail to find this, because it evaluates the inner expression first:

```
map( schedule, all-of( one-of(roomA,roomB), roomA) )
   → all-of(schedule(roomA), schedule(roomA) )
   → null [can't schedule roomA twice]
```

It is, of course, possible to avoid such problems, but only at the expense of complexity. To date, the simple depth-first search has proved satisfactory for our application.

Note also that, for the case where the mapped function has side-effects, it will be necessary to undo those if the expression fails. A typical usage pattern is:

```
result = map( function, expression )
if( result != null )
  return result
else
  undo side-effects, then return null
```

```
Simplify( expression )
{
    if expression has no elements
          return null
    else if expression has one element
          return element
    else return expression

}
map( fn, all-of( expression ))
{

   resultExpression = all-of()
      for each element in the expression
          if element is an expression
               r = map( fn, element )
          else
               r = fn(element)
          if( r == null )
               return null
          else
               include r in resultExpression
    return simplify( resultExpression )
}
map( fn, any-of( expression ))
{

   resultExpression = all-of()
      for each element in the expression
          if element is an expression
               r = map( fn, element )
          else
               r = fn(element)
          if( r != null )
               include r in resultExpression
    return simplify( resultExpression )
}
map( fn, one-of( expression ))
{

    for each element in the expression
          if element is an expression
               r = map( fn, element )
          else
               r = fn(element)
          if( r != null )
               return r
    return null

}
```

Figure 3 The recursive definition of the map function. (In practice, the implementation is a little more complicated, since we permit an additional userData parameter to be passed along with the function to be mapped).

8 References

1 F. Dawson*, et al*, iCalendar Message-based Interoperability Protocol – iMIP, RFC 2445, November 1998

2 S. Silverberg, *et al*, iCalendar Transport-independent Interoperability Protocol - iTIP, RFC 2446, November 1998

3 F. Dawson, iCalendar XML DTD, Internet Draft draft-dawson-ical-xml-dtd-01, December 1998

4 L Lippert, S. Reddy and D. Royer, iCal in XML, Internet Draft draft-reddy-xml-ical-00, November 1998

5 Foundation for Intelligent Physical Agents, FIPA ACL Message Structure Specification, document XC00061D, http://www.fipa.org, 2000

6 Foundation for Intelligent Physical Agents, FIPA SL Content Language Specification, document XC00008D, http://www.fipa.org, 2000