

Venture Beat Features Montefiore's Healthcare project with AllegroGraph

From VentureBeat August 2018

This article discusses Montefiore's PALM project that uses AllegroGraph:

Montefiore is one of the largest employers in New York State. It's also one of the busiest health care complexes – hundreds of thousands of patients pass through its sprawling campuses, which include Montefiore Medical Center, the Albert Einstein College of Medicine, and Montefiore Medical Park.

Those logistical challenges catalyzed the development of Montefiore's Patient-centered Analytical Learning Machine (PALM), a machine learning platform built from the ground up to predict and prevent life-threatening medical conditions and minimize wait times.

PALM juggles lots of datasets – electronic medical records, insurance billing codes, drug databases, and clinical trial results, to name a few. And its analytical models recently expanded to handle voice, images, and sensor inputs from internet of things devices.

Core to the semantic graph model are triplestores, which are a type of database optimized for filing away and retrieving triples. They're an entity composed of subject-predicate-object – "John has tuberculosis," for example – which PALM builds dynamically, as needed. Along the way, the system uses a frame data language, or FDL, to resolve ambiguities, like when some electronic records refer to medication by its brand instead of by its scientific name (e.g., "Advil" or "Motrin" instead of ibuprofen).

Read the full article over at Venture Beat.

Transmuting Machine Learning into Verifiable Knowledge

From AI Business – August 2018

This article covers machine learning and AI:

According to Franz CEO Jans Aasman, these machine learning deployments not only maximize organizational investments in them by driving business value, but also optimize the most prominent aspects of the data systems supporting them.

“You start with the raw data...do analytics on it, get interesting results, then you put the results of the machine learning back in the database, and suddenly you have a far more powerful database,” Aasman said.

Dr. Aasman is further quoted:

For internal applications, organizations can use machine learning concepts (such as co-occurrence—how often defined concepts occur together) alongside other analytics to monitor employee behavior, efficiency, and success with customers or certain types of customers. Aasman mentioned a project management use case for a consultancy company in which these analytics were used to “compute for every person, or every combination of persons, whether or not the project was successful: meaning, done on time to the satisfaction of the customer.”

Organizations can use whichever metrics are relevant for their businesses to qualify success. This approach is useful for determining a numerical rating for employees “and you could put that rating back in the database,” Aasman said. “Now you can do a follow up query where you say how much money did I make on the top 10 successful people; how much money did I lose on the top 10 people I don’t make a profit on.”

Read the full article over at [AI Business](#).

Using AI and Semantic Data Lakes in Healthcare – FeibusTech Research Report



Artificial intelligence has the potential to make huge improvements in just about every aspect of healthcare. Learn how Montefiore Health Systems is using semantic data lakes, architectures, and triplestores to power AI patient-centered learning. With origins in post-9/11 municipal emergency projects, Montefiore Health Systems platform – called PALM, short for patient-centered Analytical Learning Machine – is beginning to prove itself out in the Intensive Care Unit,

helping doctors save lives by flagging patients headed toward respiratory failure.

Intel and Montefiore in collaboration with FeibusTech have released a Research Brief covering Montefiore's PALM Platform (aka – The Semantic Data Lake) powered by AllegroGraph.

“Just atop all the databases is what’s known as a triplestore, or triple, construct. That’s a key piece of any semantic data architecture. A triple is a three-part data series with a common grammar structure: that is, subject-predicate-object. Like, for example, John Smith has hives. Or Jill Martin takes ibuprofen.”

“Triples are the heart and soul of graph databases, or graphs, a powerful, labor-saving approach that associates John and Jill to records of humans, hives to definitions of maladies and Ibuprofen to catalogues of drugs. And then it builds databases on the fly for the task at hand based on those associations.”

Read the full article on Intel's website to learn more about healthcare solutions based on AllegroGraph.

Navigating time in knowledge graphs

Franz's CEO, Jans Aasman, recently wrote the following article for InfoWorld.

The temporal benefits of cognitive knowledge graphs can affect almost any business problem, including basic issues of data management such as data quality, data cleansing, and integration

The concept of time presents several distinct challenges for data management, particularly as it applies to databases or stores. Those difficulties are related to the nature of time, which is ongoing, and its expressions in repositories. The former means data are relevant both at state (a point in time) and over periods of time, which increases the complexity.

Read the Full Article

The Most Secure Graph Database Available

Triples offer a way of describing model elements and relationships between them. In some cases, however, it is also convenient to be able to store data that is associated with a triple as a whole rather than with a particular element. For instance one might wish to record the source from which a triple has been imported or access level necessary to include it in query results. Traditional solutions of this problem include using graphs, RDF reification or triple IDs. All of these approaches suffer from various flexibility and performance issues. For this reason AllegroGraph offers an alternative: triple attributes.

Attributes are key-value pairs associated with a triple. Keys refer to attribute definitions that must be added to the store

before they are used. Values are strings. The set of legal values of an attribute can be constrained by the definition of that attribute. It is possible to associate multiple values of a given attribute with a single triple.

Possible uses for triple attributes include:

- *Access control: It is possible to instruct AllegroGraph to prevent an user from accessing triples with certain attributes.*
- *Sharding: Attributes can be used to ensure that related triples are always placed in the same shard when AllegroGraph acts as a distributed triple store.*

Like all other triple components, attribute values are immutable. They must be provided when the triple is added to the store and cannot be changed or removed later.

To illustrate the use of triple attributes we will construct an artificial data set containing a log of information about contacts detected by a submarine at a single moment in time.

Managing attribute definitions

Before we can add triples with attributes to the store we must create appropriate attribute definitions.

First let's open a connection

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

Attribute definitions are represented by **AttributeDefinition** objects. Each definition has a name, which must be unique, and a few optional properties (that can also be passed as constructor arguments):

- **allowed_values**: a list of strings. If this property is set then only the values from this list can be used for the defined attribute.
- **ordered**: a boolean. If true then attribute value comparisons will use the ordering defined by **allowed_values**. The default is false.
- **minimum_number**, **maximum_number**: integers that can be used to constrain the cardinality of an attribute. By default there are no limits.

Let's define a few attributes that we will later use to demonstrate various attribute-related capabilities of AllegroGraph. To do this, we will use the `setAttributeDefinition()` method of the connection object.

```
from franz.openrdf.repository.attributes import AttributeDefinition

# A simple attribute with no constraints governing the set
# of legal values or the number of values that can be
# associated with a triple.
tag = AttributeDefinition(name='tag')

# An attribute with a limited set of legal values.
# Every bit of data can come from multiple sources.
# We encode this information in triple attributes,
# since it refers to the tripe as a whole. Another
# way of achieving this would be to use triple ids
# or RDF reification.
source = AttributeDefinition(
    name='source',
    allowed_values=['sonar', 'radar', 'esm', 'visual'])

# Security level - notice that the values are ordered
# and each triple must have exactly one value for
# this attribute. We will use this to prevent some
# users from accessing classified data.
level = AttributeDefinition(
    name='level',
    allowed_values=['low', 'medium', 'high'],
    ordered=True,
    minimum_number=1,
    maximum_number=1)

# An attribute like this could be used for sharding.
# That would ensure that data related to a particular
# contact is never partitioned across multiple shards.
# Note that this attribute is required, since without
# it an attribute-sharded triple store would not know
# what to do with a triple.
contact = AttributeDefinition(
    name='contact',
    minimum_number=1,
    maximum_number=1)

# So far we have created definition objects, but we
# have not yet sent those definitions to the server.
# Let's do this now.
conn.setAttributeDefinition(tag)
conn.setAttributeDefinition(source)
```

```
conn.setAttributeDefinition(level)
conn.setAttributeDefinition(contact)

# This line is not strictly necessary, because our
# connection operates in autocommit mode.
# However, it is important to note that attribute
# definitions have to be committed before they can
# be used by other sessions.
conn.commit()
```

It is possible to retrieve the list of attribute definitions from a repository by using the **getAttributeDefinitions()** method:

```
for attr in conn.getAttributeDefinitions():
    print('Name: {0}'.format(attr.name))
    if attr.allowed_values:
        print('Allowed values: {0}'.format(
            ', '.join(attr.allowed_values)))
        print('Ordered: {0}'.format(
            'Y' if attr.ordered else 'N'))
    print('Min count: {0}'.format(attr.minimum_number))
    print('Max count: {0}'.format(attr.maximum_number))
    print()
```

Notice that in cases where the maximum cardinality has not been explicitly defined, the server replaced it with a default value. In practice this value is high enough to be interpreted as 'no limit'.

```
Name: tag
Min count: 0
Max count: 1152921504606846975

Name: source
Allowed values: sonar, radar, esm, visual
Min count: 0
Max count: 1152921504606846975
Ordered: N

Name: level
Allowed values: low, medium, high
Ordered: Y
Min count: 1
Max count: 1

Name: contact
Min count: 1
Max count: 1
```


Attribute definitions can be removed (provided that the attribute is not used by the static attribute filter, which will be discussed later) by calling **deleteAttributeDefinition()**:

```
conn.deleteAttributeDefinition('tag')
defs = conn.getAttributeDefinitions()
print(', '.join(sorted(a.name for a in defs)))
```

```
contact, level, source
```

Adding triples with attributes

Now that the attribute definitions have been established we can demonstrate the process of adding triples with attributes. This can be achieved using various methods. A common element of all these methods is the way in which triple attributes are represented. In all cases dictionaries with attribute names as keys and strings or lists of strings as values are used.

When **addTriple()** is used it is possible to pass attributes in a keyword parameter, as shown below:

```
ex = conn.namespace('ex://')
conn.addTriple(ex.S1, ex.cls, ex.Udaloy, attributes={
    'source': 'sonar',
    'level': 'low',
    'contact': 'S1'
})
```

The **addStatement()** method works in similar way. Note that it is not possible to include attributes in the **Statement** object itself.

```
from franz.openrdf.model import Statement

s = Statement(ex.M1, ex.cls, ex.Zumwalt)
conn.addStatement(s, attributes={
    'source': ['sonar', 'esm'],
    'level': 'medium',
    'contact': 'M1'
})
```

When adding multiple triples with **addTriples()** one can add a fifth element to each tuple to represent attributes. Let us illustrate

this by adding an aircraft to our dataset.

```
conn.addTriples([
    (ex.R1, ex.cls, ex['Ka-27'], None,
     {'source': 'radar',
      'level': 'low',
      'contact': 'R1'}),
    (ex.R1, ex.altitude, 200, None,
     {'source': 'radar',
      'level': 'medium',
      'contact': 'R1'})])
```

When all or most of the added triples share the same attribute set it might be convenient to use the `attributes` keyword parameter. This provides default values, but is completely ignored for all tuples that already contain attributes (the dictionaries are not merged). In the example below we add a triple representing an aircraft carrier and a few more triples that specify its position. Notice that the first triple has a lower security level and multiple sources. The common `'contact'` attribute could be used to ensure that all this data will remain on a single shard.

```
conn.addTriples([
    (ex.M2, ex.cls, ex.Kuznetsov, None, {
        'source': ['sonar', 'radar', 'visual'],
        'contact': 'M2',
        'level': 'low',
    }),
    (ex.M2, ex.position, ex.pos343),
    (ex.pos343, ex.x, 430.0),
    (ex.pos343, ex.y, 240.0)],
    attributes={
        'contact': 'M2',
        'source': 'radar',
        'level': 'medium'
    })
```

Another method of adding triples with attributes is to use the NQX file format. This works both with `addFile()` and `addData()` (illustrated below):

```
from franz.openrdf.rio.rdfformat import RDFFormat

conn.addData('''
    <ex://S2> <ex://cls> <ex://Alpha> \N
    {"source": "sonar", "level": "medium", "contact": "S2"} .
    <ex://S2> <ex://depth> "300" \N
```

```

{"source": "sonar", "level": "medium", "contact": "S2"} .
<ex://S2> <ex://speed_kn> "15.0" \
{"source": "sonar", "level": "medium", "contact": "S2"} .
''' , rdf_format=RDFFormat.NQX)

```

When importing from a format that does not support attributes, it is possible to provide a common set of attribute values with a keyword parameter:

```

from franz.openrdf.rio.rdfformat import RDFFormat

conn.addData('''
    <ex://V1> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 100 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V2> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 200 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V3> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 300 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V4> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 400 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V5> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 500 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V6> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 600 ;
              <ex://speed_kn> 12.0e+8 .
''', attributes={
    'source': 'visual',
    'level': 'high',
    'contact': 'a therapist'})

```

The data above represents six visually observed Walrus-class submarines, flying at different altitudes and well above the speed of light. It has been highly classified to conceal the fact that someone has clearly been drinking while on duty – after all there are only four Walrus-class submarines currently in service, so the observation is obviously incorrect.

Retrieving attribute values

We will now print all the data we have added to the store, including attributes, to verify that everything worked as

expected. The only way to do that is through a SPARQL query using the appropriate [magic property](#) to access the attributes. The query below binds a literal containing a JSON representation of triple attributes to the `?a` variable:

```
import json

r = conn.executeTupleQuery('''
    PREFIX attr: <http://franz.com/ns/allegrograph/6.2.0/>
    SELECT ?s ?p ?o ?a {
        ?s ?p ?o .
        ?a attr:attributes (?s ?p ?o) .
    } ORDER BY ?s ?p ?o''')
with r:
    for row in r:
        print(row['s'], row['p'], row['o'])
        print(json.dumps(json.loads(row['a'].label),
                          sort_keys=True,
                          indent=4))
```

The result contains all the expected triples with pretty-printed attributes.

```
<ex://M1> <ex://cls> <ex://Zumwalt>
{
  "contact": "M1",
  "level": "medium",
  "source": [
    "esm",
    "sonar"
  ]
}
<ex://M2> <ex://cls> <ex://Kuznetsov>
{
  "contact": "M2",
  "level": "low",
  "source": [
    "visual",
    "radar",
    "sonar"
  ]
}
<ex://M2> <ex://position> <ex://pos343>
{
  "contact": "M2",
  "level": "medium",
  "source": "radar"
}
```

```
<ex://R1> <ex://altitude> "200"^^...
{
  "contact": "R1",
  "level": "medium",
  "source": "radar"
}
<ex://R1> <ex://cls> <ex://Ka-27>
{
  "contact": "R1",
  "level": "low",
  "source": "radar"
}
<ex://S1> <ex://cls> <ex://Udaloy>
{
  "contact": "S1",
  "level": "low",
  "source": "sonar"
}
<ex://S2> <ex://cls> <ex://Alpha>
{
  "contact": "S2",
  "level": "medium",
  "source": "sonar"
}
<ex://S2> <ex://depth> "300"
{
  "contact": "S2",
  "level": "medium",
  "source": "sonar"
}
<ex://S2> <ex://speed_kn> "15.0"
{
  "contact": "S2",
  "level": "medium",
  "source": "sonar"
}
<ex://V1> <ex://altitude> "100"^^...
{
  "contact": "a therapist",
  "level": "high",
  "source": "visual"
}
<ex://V1> <ex://cls> <ex://Walrus>
{
  "contact": "a therapist",
  "level": "high",
  "source": "visual"
}
<ex://V1> <ex://speed_kn> "1.2E9"^^...
{
```

```

    "contact": "a therapist",
    "level": "high",
    "source": "visual"
  }
  ...
<ex://pos343> <ex://x> "4.3E2"^^...
{
  "contact": "M2",
  "level": "medium",
  "source": "radar"
}
<ex://pos343> <ex://y> "2.4E2"^^...
{
  "contact": "M2",
  "level": "medium",
  "source": "radar"
}

```

Attribute filters

Triple attributes can be used to provide fine-grained access control. This can be achieved by using [static attribute filters](#).

Static attribute filters are simple expressions that control which triples are visible to a query based on triple attributes. Each repository has a single, global attribute filter that can be modified using **setAttributeFilter()**. The values passed to this method must be either strings (the syntax is described in the documentation of [static attribute filters](#)) or filter objects.

Filter objects are created by applying set operators to ‘attribute sets’. These can then be combined using filter operators.

An attribute set can be one of the following:

- *a string or a list of strings: represents a constant set of values.*
- *TripleAttribute.name: represents the value of the name attribute associated with the currently inspected triple.*
- *UserAttribute.name: represents the value of the name attribute associated with current query. User attributes will be discussed in more detail later.*

Available set operators are shown in the table below. All classes and functions mentioned here can be imported from the `franz.openrdf.repository.attributes` package:

Syntax	Meaning
<code>Empty(x)</code>	True if the specified attribute set is empty.
<code>Overlap(x, y)</code>	True if there is at least one matching value between the two attribute sets.
<code>Subset(x, y), x << y</code>	True if every element of <i>x</i> can be found in <i>y</i>
<code>Superset(x, y), x >> y</code>	True if every element of <i>y</i> can be found in <i>x</i>
<code>Equal(x, y), x == y</code>	True if <i>x</i> and <i>y</i> have exactly the same contents.
<code>Lt(x, y), x < y</code>	True if both sets are singletons, at least one of the sets refers to a triple or user attribute, the attribute is ordered and the value of the single element of <i>x</i> occurs before the single value of <i>y</i> in the <code>lowed_values</code> list of the attribute.
<code>Le(x, y), x <= y</code>	True if <i>y</i> < <i>x</i> is false.
<code>Eq(x, y)</code>	True if both <i>x</i> < <i>y</i> and <i>y</i> < <i>x</i> are false. Note that using the <code>==</code> Python operator translates to <i>Eqauls</i> , not <i>Eq</i> .
<code>Ge(x, y), x >= y</code>	True if <i>x</i> < <i>y</i> is false.
<code>Gt(x, y), x > y</code>	True if <i>y</i> < <i>x</i> .

Note that the overloaded operators only work if at least one of the attribute sets is a `UserAttribute` or `TripleAttribute` reference – if both arguments are strings or lists of strings the default Python semantics for each operator are used. The prefix syntax always produces filters.

Filters can be combined using the following operators:

Syntax	Meaning
<code>Not(x), ~x</code>	Negates the meaning of the filter.
<code>And(x, y, ...), x & y</code>	True if all subfilters are true.
<code>Or(x, y, ...), x y</code>	True if at least one subfilter is true.

Filter operators also work with raw strings, but overloaded operators will only be recognized if at least one argument is a filter object.

Using filters and user attributes

The example below displays all classes of vessels from the dataset after establishing a static attribute filter which ensures that only sonar contacts are visible:

```
from franz.openrdf.repository.attributes import *

conn.setAttributeFilter(TripleAttribute.source >> 'sonar')
conn.executeTupleQuery(
    'select ?class { ?s <ex://cls> ?class } order by ?class',
    output=True)
```

The output contains neither the visually observed Walruses nor the radar detected ASW helicopter.

```
-----
| class          |
=====
| ex://Alpha     |
| ex://Kuznetsov |
| ex://Udaloy    |
| ex://Zumwalt   |
-----
```

To avoid having to set a static filter before each query (which would be inefficient and cause concurrency issues) we can employ user attributes. User attributes are specific to a particular connection and are sent to the server with each query. The static attribute filter can refer to these and compare them with triple attributes. Thus we can use code presented below to create a filter which ensures that a connection only accesses data at or

below the chosen clearance level.

```
conn.setUserAttributes({'level': 'low'})
conn.setAttributeFilter(
    TripleAttribute.level <= UserAttribute.level)
conn.executeTupleQuery(
    'select ?class { ?s <ex://cls> ?class } order by ?class',
    output=True)
```

We can see that the output here contains only contacts with the access level of *low*. It omits the destroyer and Alpha submarine (these require *medium* level) as well as the top-secret Walruses.

```
-----
| class          |
=====
| ex://Ka-27     |
| ex://Kuznetsov |
| ex://Udaloy    |
-----
```

The main advantage of the code presented above is that the filter can be set globally during the application setup and access control can then be achieved by varying user attributes on connection objects.

Let us now remove the attribute filter to prevent it from interfering with other examples. We will use the **clearAttributeFilter()** method.

```
conn.clearAttributeFilter()
```

It might be useful to change connection's attributes temporarily for the duration of a single code block and restore prior attributes after that. This can be achieved using the **temporaryUserAttributes()** method, which returns a context manager. The example below illustrates its use. It also shows how to use **getUserAttributes()** to inspect user attributes.

```
with conn.temporaryUserAttributes({'level': 'high'}):
    print('User attributes inside the block:')
    for k, v in conn.getUserAttributes().items():
        print('{0}: {1}'.format(k, v))
    print()
print('User attributes outside the block:')
for k, v in conn.getUserAttributes().items():
    print('{0}: {1}'.format(k, v))
```

```
User attributes inside the block:  
level: high
```

```
User attributes outside the block:  
level: low »
```

Allegro Knowledge Graph News

Franz periodically distributes newsletters to its Semantic Technologies, and Common Lisp based Enterprise Development Tools mailing lists, providing information on related upcoming events and new software product developments.

Read our latest AllegroGraph newsletter.

Previous issues are listed in the Newsletter Archive.

Harmonizing big data with an enterprise knowledge graph

Franz's CEO, Jans Aasman, recently wrote the following article for InfoWorld.



In addition to streamlining how users retrieve diverse data via automation capabilities, a knowledge graph standardizes

One of the most significant results of the big data era is the broadening diversity of data types required to solidify data as an enterprise asset. The maturation of technologies addressing scale and speed has done little to decrease the difficulties associated with complexity, schema transformation and integration of data necessary for informed action.

The influence of cloud computing, mobile technologies, and distributed computing environments contribute to today's variegated IT landscape for big data. Conventional approaches to master data management and data lakes lack critical requirements to unite data—regardless of location—across the enterprise for singular control over multiple sources.

The enterprise knowledge graph concept directly addresses these limitations, heralding an evolutionary leap forward in big data management. It provides singular access for data across the enterprise in any form, harmonizes those data in a standardized format, and assists with the facilitation of action required to repeatedly leverage them for use cases spanning organizations and verticals.

Semantic Computing, Predictive Analytics Need

Reliable Metadata



Our Healthcare Partners at Montefiore were interviewed at Health Analytics:

Reliable metadata is the key to leveraging semantic computing and predictive analytics for healthcare applications, such as population health management and crisis care.

As the healthcare industry reaches the saturation point of electronic health record adoption, and slowly moves past the pain of the implementation process, it may seem like the right time to stop thinking so much about hammering home basic data governance principles for staff members and start looking at the next phase of health IT implementation: the big data analytics environment.

After all, most providers are now sitting on an enormous nest egg of patient data, which may be just clean, complete, and standardized enough to start experimenting with population health management, operational analytics, or even a bit of predictive risk stratification. Many healthcare organizations are experimenting with these advanced analytics projects in an effort to prepare themselves for the financial storm that is approaching with the advent of value-based care.

The immense pressure to cut costs, meet quality benchmarks, shoulder financial risk, and improve patient outcomes is causing no small degree of anxiety for providers, who are racing to batten down the hatches before the typhoon overtakes them.

While it may be tempting to jump into quick-win analytics that use “good enough” datasets to solve a specific pressing use case, providers may be at risk of repeating the same

mistakes they made with slapdash EHR implementations: creating data siloes, orphaned reports, and poor quality datasets that cannot be used in a reliable, repeatable way for meaningful quality improvements.

Read the full article at [Health Analytics](#)

Montefiore Semantic Data Lake Tackles Predictive Analytics

Montefiore Medical Center is preparing to launch a sophisticated predictive analytics program for crisis patients, which is rooted in its real-time semantic data lake technology.

Semantic computing is becoming a hot topic in the healthcare industry as the first wave of big data analytics leaders looks to move beyond the basics of population health management, predictive analytics, and risk stratification.

This new approach to analytics eschews the rigid, limited capabilities of the traditional relational database and instead focuses on creating a fluid pool of standardized data elements that can be mixed and matched on the fly to answer a large number of unique queries.

Montefiore Medical Center, in partnership with Franz Inc., is among the first healthcare organizations to invest in a robust semantic data lake as the foundation for advanced clinical

decision support and predictive analytics capabilities.

Read the full article at [Health IT Analytics](#)

AllegroGraph News

Franz periodically distributes newsletters to its Semantic Technologies, and Common Lisp based Enterprise Development Tools mailing lists, providing information on related upcoming events and new software product developments.