ALIVE:
Monitoring Architecture Draft

Ignasi Gómez-Sebastià (igomez@lsi.upc.edu)
Sergio Alvarez-Napagao (salvarez@lsi.upc.edu)

Table of contents

1 Introduction

This purpose of this document is to describe the expected monitoring capabilities, procedures and components for the ALIVE architecture, in order to fulfil the requirements for monitoring presented in the architecture document.

1.1 TODO

This is a draft document, and thus it is still incomplete. The following list defines tasks that have been identified for incoming changes:

- Add references to other documents/papers
- Global Monitor section
- Event Bus, Event Log descriptions
- Monitor Tool section

1.2 Change Log

27.07.2009 - Added Monitor Metamodel
- Updated Event Metamodel
- Added description about differences between Local Monitor and Global - Monitor
- Update Figure 1 with the latest version

2 Requirements from the architecture

The architecture document introduces some requirements that have to be implemented by the monitoring architecture proposed in the present document. This section summarizes these requirements.

From the global architecture diagram, there are several components and subcomponents that are directly involved with the monitoring process. They are identified (in red) in

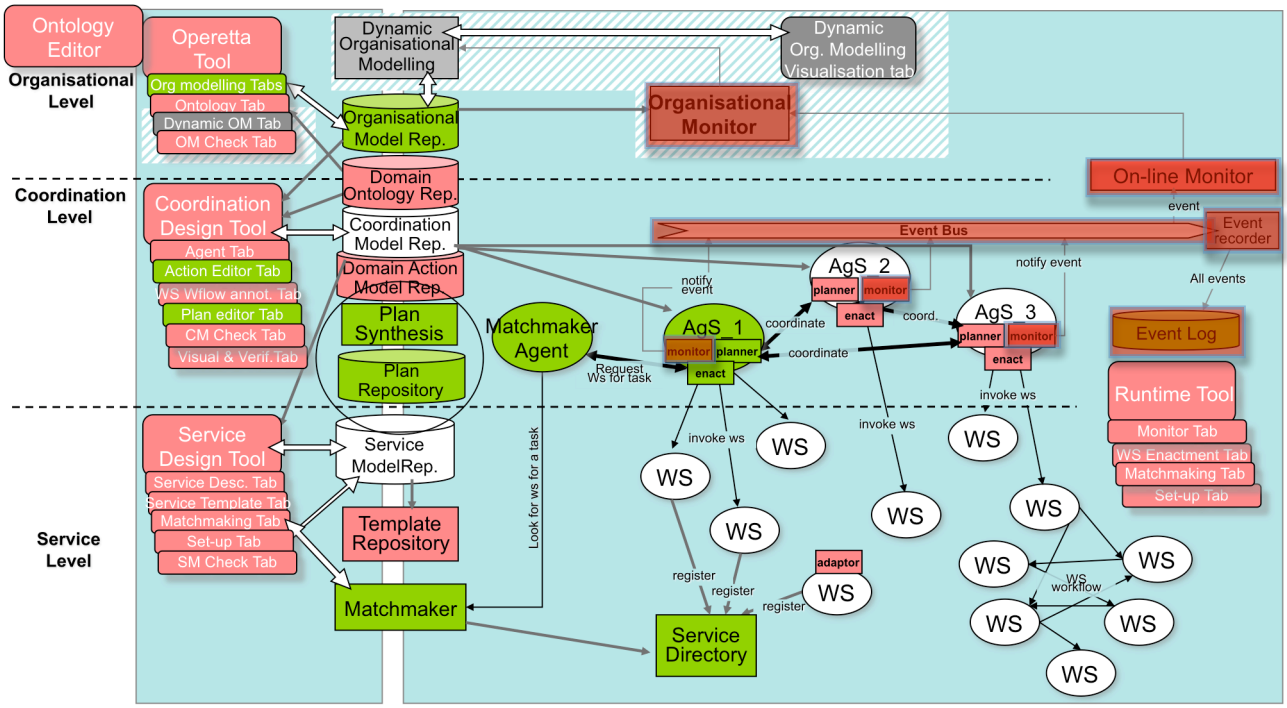


Figure 1. Monitoring components in the ALIVE architecture

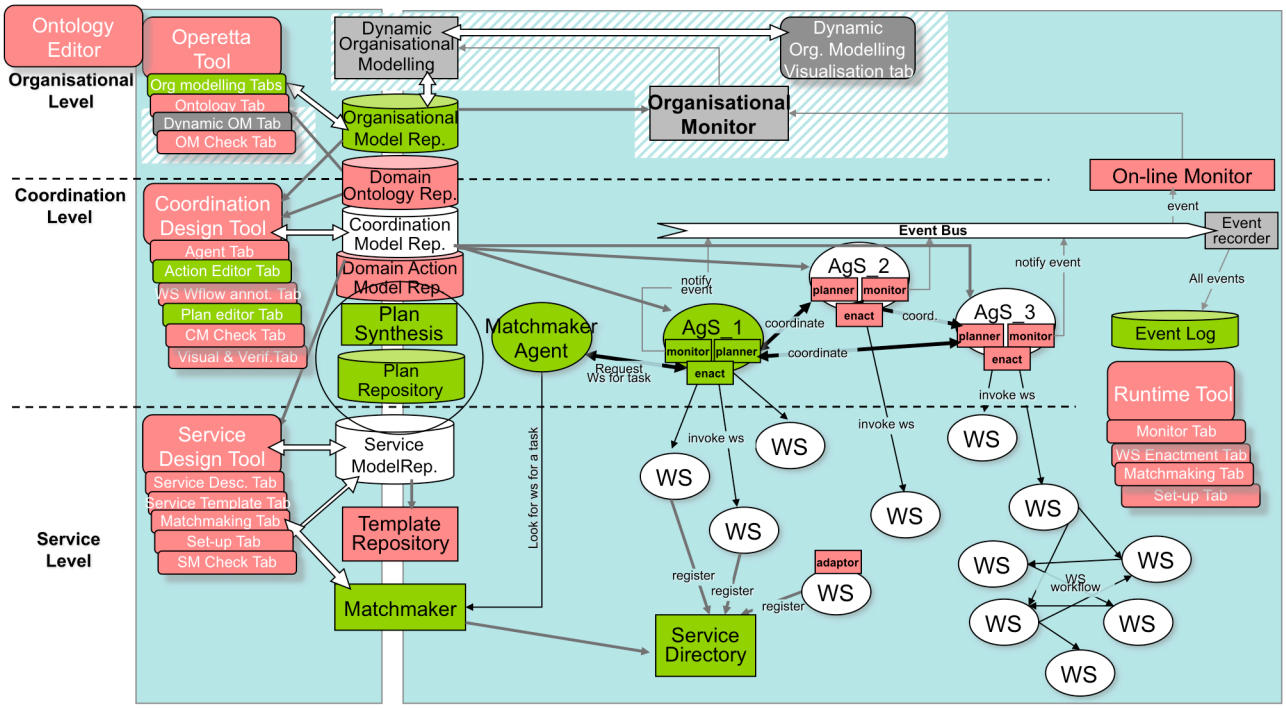


Figure 1.

In the following subsections we introduce the elements that will be described through the rest of this document.

2.1 Events

Events are the atomic element the monitoring deals with. For the purpose of this document, we will consider as an event any *happening* of *relevance* that occurs within the *observable boundaries* of the system, where:

- *A happening* is a computable item: a change of state, or a fact related to a resource or an action;
- *Relevance* means that a specific item should be taken into consideration for a proper fulfilment of goals, objectives, or normative compliance; and
- The *Observable boundaries* of a system is the total set of items that can be "seen", limited by the technical and/or physical capabilities of the actors of the system.

Events should ideally be described by means of an ontology, in order to be understandable by the actors processing them. In ALIVE, being a layered architecture, there will be events representing facts at different levels of abstraction.

However, it is our objective to design the monitoring architecture in such a way that events are seen as abstract objects. For this reason, we will consider events as abstract objects, independent of the information they carry. It will be the responsibility of the actor processing an event to interpret the contents and the context of those contents.

2.2 Monitor

The ALIVE architecture is expected to provide on-line monitoring mechanisms to detect failures or deviations from expected function and the mechanisms to correct, recover or adapt to them. These services are to be packed into a single high level component called *Monitor*. This component is to be designed from an abstract point of view, to be instantiated at different levels of scope.

The *Monitor* component is the instantiation of the Service Monitoring framework (defined in the ALIVE Architecture document). It is able to aggregate and analyse events related to the execution of services, the fulfilment of some coordination plans and the achievement of some role and/or organisational objectives. During the on-line execution events are generated by the components in agentified services, whenever deviations, exceptions or failures are detected that cannot be handled by the agentified service itself or the existing coordination plan in place. In such situations the current Organisational Model is evaluated and then either (a) the objectives affected by the detected issue may be re-evaluated (its priority lowered down or even dropped completely), or (b) a more deep change in the Organisation Model may be required (for instance, changing the rights of a role).

There will be at least two instantiations: the *Local Monitor* and the *Global Monitor*.

2.2.1 Local Monitor

The *Local Monitor* is a local component, available to the agents. In order to fulfil the objectives of a specific agent, this component may be optionally used.

Its responsibilities are limited to the translation of monitored raw events to a higher level of abstraction understandable by the Organisational Level components.

2.2.2 Global Monitor

The *Global Monitor* is a global component, used at the Organisational Level. It will have knowledge of the organisational domain, and will process the (preprocessed) events produced by the Local Monitors in order to detect violations of the organisational norms and act accordingly.

In this sense, the responsibilities of a global monitor are broader than those of local monitors, as global monitors will make decisions about how to assign sanctions and create repair plans for violation states. In some cases, global monitors may even execute actions themselves.

2.3 Monitor Tool

The *Monitor Tool* is a graphical interface, which allows the administrator to inspect the status of a system's execution, getting information from the different components in the on-line architecture (especially from the Monitor one). In this way the administrator can keep track of the events that are generated through execution and inspect how the system handles them.

3 Events

The *event metamodel* is shown in Figure 2. As explained in Section 2.1, our objective is to model an event in the most possible general meaning of the concept, while representing additional attributes that may help not only understand the fact, but also unequivocally identify it and place the event in both time and location.

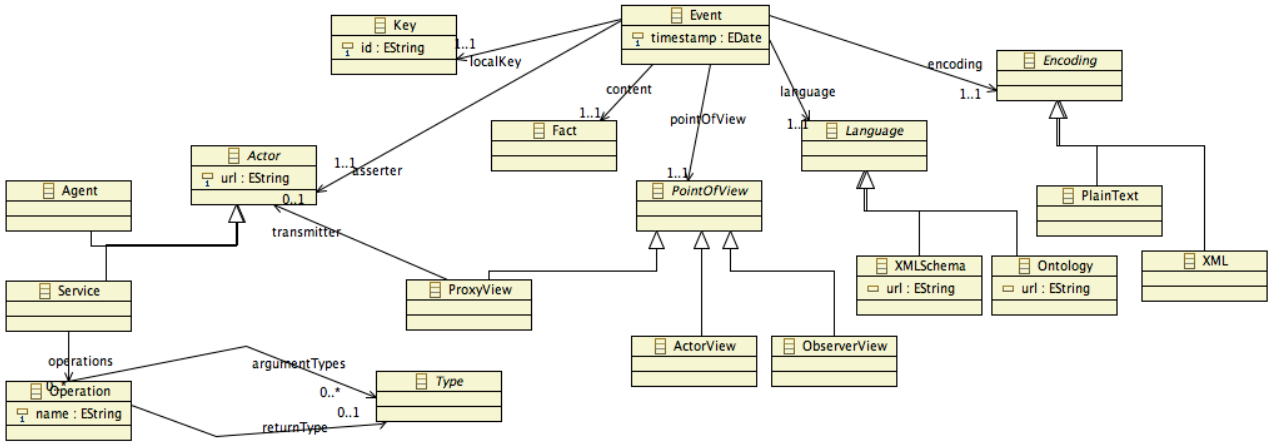


Figure 2. Event metamodel

The main element, *Event*, represents the concept of event itself. It includes the following attributes:

- The *content*, which is the central attribute, and can be any kind of object.
- The *encoding*, which defines how the content is represented, e.g. as a plain text string, as an encrypted string (and thus defining as well how to decrypt it), or as an XML object (and thus defining which is the XSD it is defined upon).
- The *language*, defining how to interpret the encoded content. In most cases the language will be an ontology.
- The *asserter*, that is, the actor that stated the content of the event, e.g. an agent, a service, or an architectural component.
- The *local key*, which is a unique key created by the asserter. The combination of the asserter plus the local key has to be unique in the system.
- The *point of view* of the asserter with respect to the asserted fact. This attribute should describe how the asserter got conscience of the fact. In most cases, this point of view will be one of the following: the asserter was the direct responsible of the happening, the asserter observed the happening, or the asserter acts as a proxy or re-transmitter of a third party responsible for the fact.
- The *timestamp* identifying the exact point in time in which the asserter became aware of the fact.

4 Monitor

Regardless of whether a monitor is a local monitor or a global monitor, there are common properties that it has to implement.

First of all, a monitor may execute a set of rules to trigger events, e.g. send a notification of the reception of a certain event. These rules define the behaviour of a specific instance of a monitor. Our proposal is to use a *rule engine*, e.g. *Drools*, to manage and execute these rules.

The definition of these rules is based on facts to be matched, and thus a set of rules implicitly defines a set of facts to be kept under surveillance. Each monitor is then responsible for extracting this set of *relevant* facts and for subscribing them to the Event Bus.

Although a monitor may be configured to communicate directly with actors or components, as we have seen monitors are also possible providers of events. Therefore, every monitor in the system is connected to the Event Bus.

Finally, every monitor uses one or more *interface* to communicate with actors and with external or internal components. Considering that a monitor is not only a receiver but also a sender of events, these interfaces have to work in both ways. Our proposal is to use a *subscribe* mechanism for each interface.

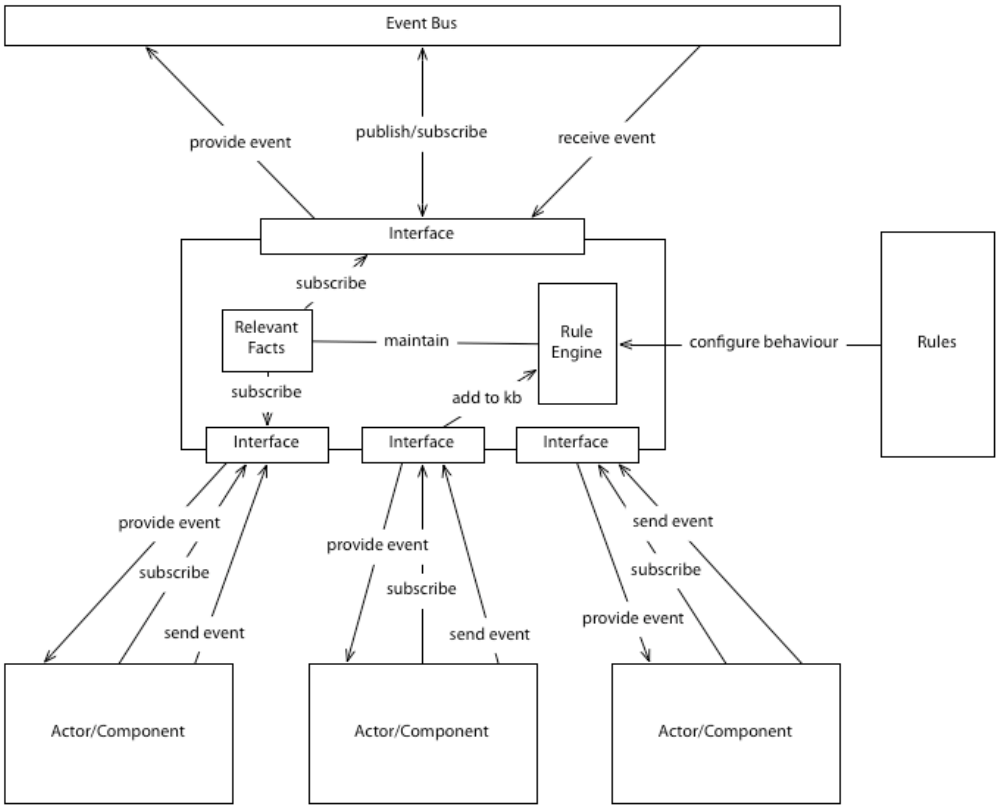


Figure 3. Monitor architecture

Figure 3 presents the generic monitor architecture, taking into account the constraints exposed in the previous paragraphs.

Regardless of the specific implementation details for each type of interface, we can define a generic API to be exposed for every actor or component that will interact with a monitor (see Figure 4).

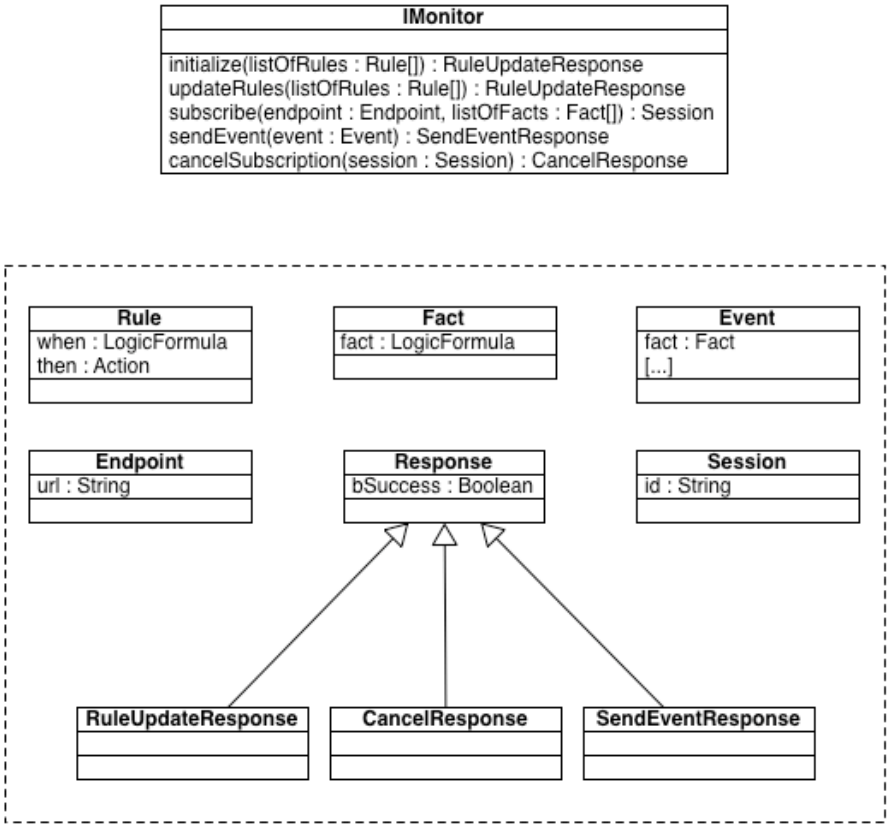


Figure 4. Monitor generic interface

initialize and *updateRules* are methods to be called by the component responsible for creating and configuring the monitor instance. These two methods will define the active set of rules and thus determining the behaviour of the monitor.

subscribe is a method public to any component willing to receive certain notifications from the monitor, by specifying a list of facts and an endpoint. This endpoint will be used by the monitor to asynchronously send the subscribed events back. The specific way of handling these notifications will depend on the implementation of the interface. This method returns a session representing the subscription.

sendEvent can be used by a component to notify an event to the monitor.

cancelSubscription can be used by a component to remove the correspondent subscription and therefore indicating the wish to stop receiving notifications.

Given the features and interfaces defined for the monitoring system, the monitoring metamodel proposed is depicted in Figure 5.

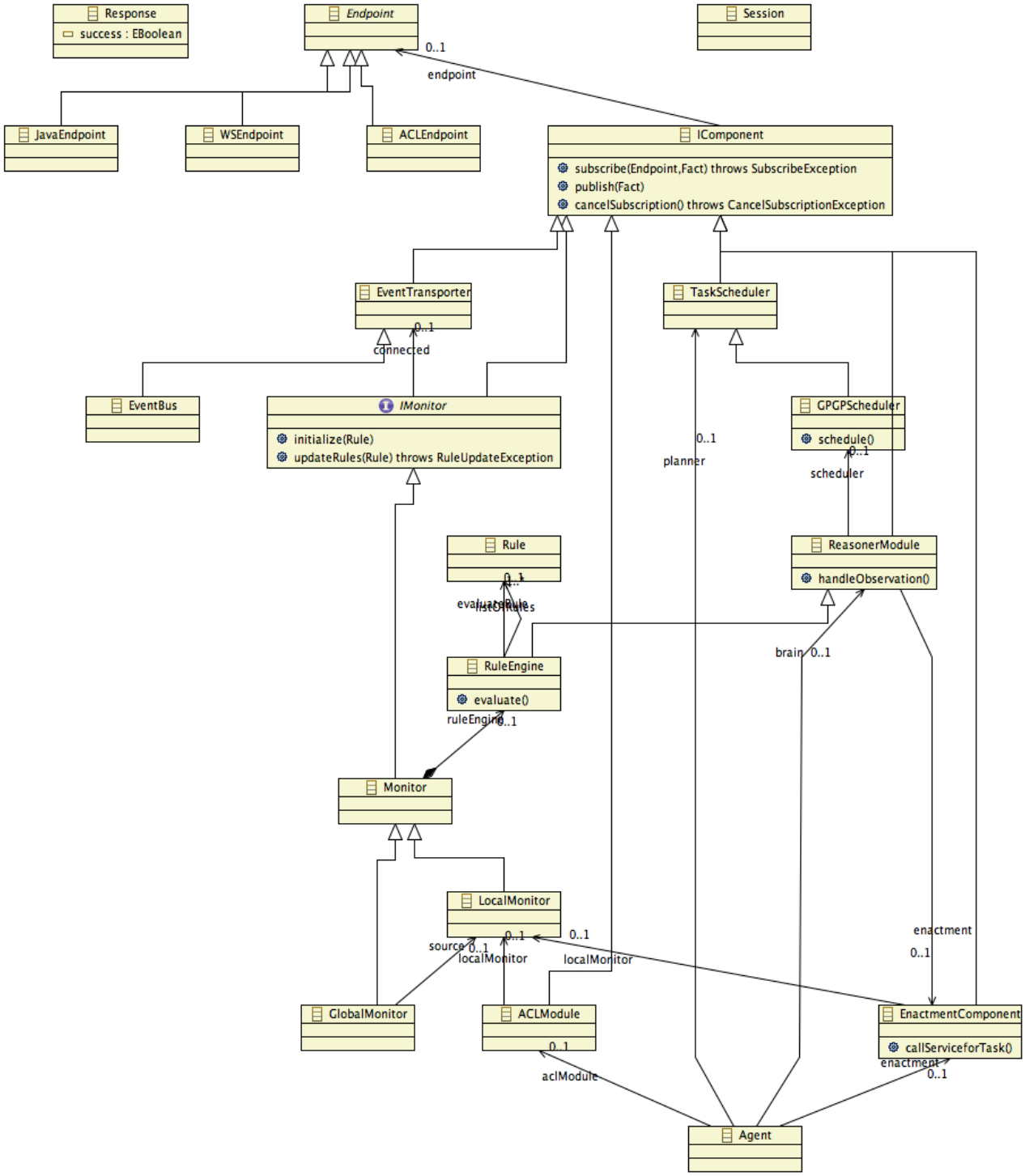


Figure 5. Monitoring metamodel

5 Local Monitor

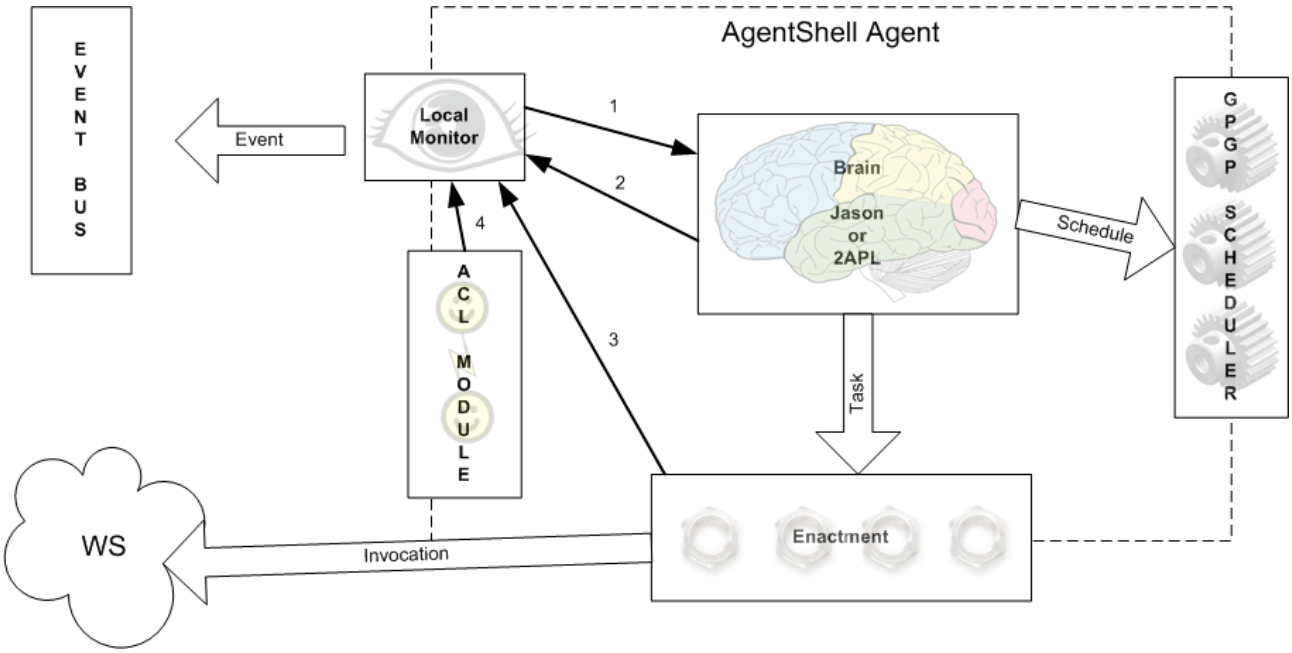


Figure 6. Connections between the Local Monitor and other components

The connections (see Figure 6) between the local monitor, and the different components inside the agent, are depicted in the following list. A proposal to connect each of these components with the local monitor is made in subsequent sections.

1. Local monitor submits observations to brain. These observations can trigger the executions of new plans, which might require the use of another components (ACL, Enactment or Scheduler).
2. The brain submits observations to the local monitor. Such observations can include the fulfilment of a given plan, or the intention to execute a given task via the enactment component.
3. The enactment submits observations to the local monitor.
4. The ACL module submits observations to the local monitor. Typically, these observations regard to the interchange (either sending or receiving) of messages between agents.
5. Scheduler notifies to the monitor plan fulfilment, plan failure, or the fact of allocating a given task to an agent.

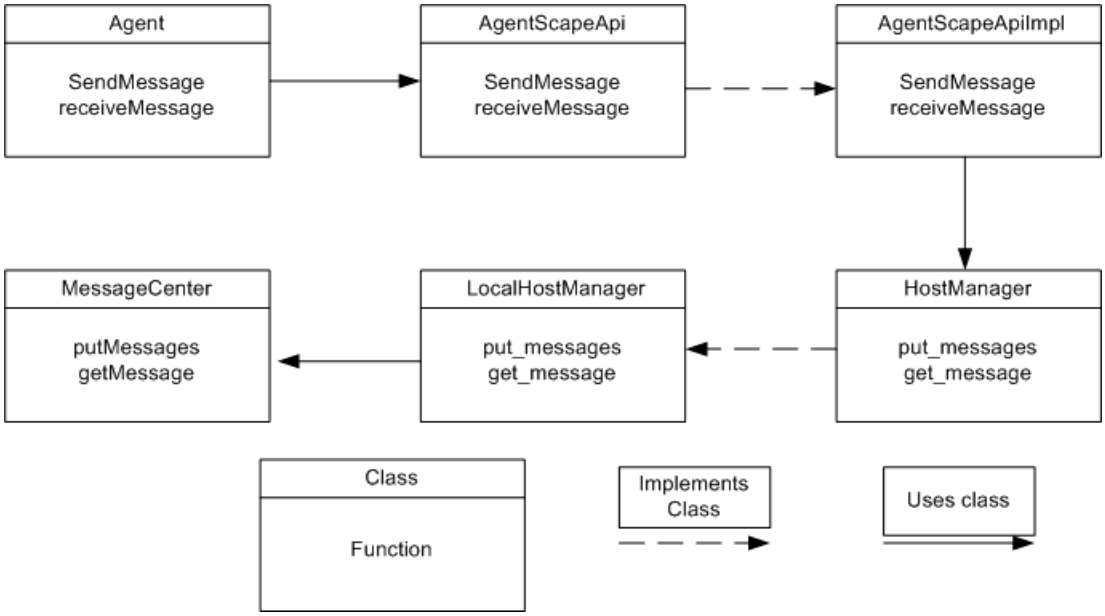
5.1 Enactment/Brain-Monitor and Monitor-Brain connection

This connection will be available via a direct Java class implementation of the *IMonitor* interface described in Section 4.

5.2 ACL-Monitor connection

The code flow used by AgentScape when an agent interchanges messages with another agent (either

sending or receiving them) is as follows:



5.2.1 Class Agent

Agent creates an envelope and sends it using function '*sendMessage*'. Agent can also receive envelopes via '*receiveMessage*' function.

```
// create an envelope from-to our primary handle
Envelope env = new Envelope (handle, handle, "Hello");

// we can now receive the same message from the primary handle
// (because it was also the target handle of the envelope)
try
{
    sendMessage (env);
    Envelope in = receiveMessage (true);

    String msg = (String) in.getData();
```

5.2.2 Class AgentScapeApi

This class is just an interface. It provides interfaces for both '*SendMessage*' and '*receiveMessage*' functions.

5.2.3 Class AgentScapeApiImpl

This class is an implementation of class '*AgentScapeApi*'. There are two more available implementations as well, '*AgentScapeApiRmiServer*' and '*AgentScapeApiSunrpcDispatcher*' but both them end up invoking the methods on '*AgentScapeApi'*.

The implementation derives functions '*SendMessage*' and '*receiveMessage*' to the invokation of 3 functions:

- put_messages: Invoked either when a set of messages is to be sent or when a single message is to be sent. In the last case a set of messages containing only one element is sent.
- get_message: Invoked when a single message is to be received
- get_messages: Invoked when several messages are to be retrieved

```
public void sendMessage (Envelope envelope)
    throws MessageException
{

    sendMessages (Arrays.asList(envelope));
}


public void sendMessages (List<Envelope> envelopes)
    throws MessageException
{
HostManager mc = null;
try
{
    mc = lookup.getMessageCenter (globalFromId);
    mc.put_messages( toSend, true );
}
```

```
public Envelope receiveMessage (AgentHandle handle,
        List<AgentHandle> handles, boolean block)
    throws AgentUnknownException
{
    AgentID ownerId = lookup.handleToAgentID( handle );
    log.debug ("Handle " + handle + " maps to id " + ownerId);

    HostManager mc = null;
    try {
        mc = lookup.getMessageCenter( ownerId );
        return mc.get_message( ownerId, handles, block );
    } catch (AgentIDUnknownException e) {
        throw new AgentUnknownException (e.getMessage(), e, handle);
    } finally {
        lookup.releaseSystemService( mc );
    }
}
```

```
public Envelope[] receiveMessages (AgentHandle handle,
        List<AgentHandle> handles, boolean block)
    throws AgentUnknownException
{
    AgentID ownerId = lookup.handleToAgentID( handle );

    HostManager mc = null;
    try {
        mc = lookup.getMessageCenter( ownerId );
        List<Envelope> result = mc.get_messages( ownerId, handles, block );
        if (result != null)
            return result.toArray( new Envelope[result.size()]);
        return new Envelope[0];
    } catch (AgentIDUnknownException e) {
        throw new AgentUnknownException (e.getMessage(), handle);
    } finally {
        lookup.releaseSystemService( mc );
    }
}
```

This class is not good to integrate AgentScape's ACL module with the local monitor because this class maps also to two another implementations of the same interface (that is '*AgentScapeApi*').

5.2.4 Class HostManager

This class is just an interface. It provides interfaces for '*put_messages*' , '*get_message*' and '*get_messages'* functions.

5.2.5 Class LocalHostManager

This class is an implementation of class '*HostManager*'. This is the only available implementation of the interface class.
The class has implementation for functions '*put_messages*' , '*get_message*' and '*get_messages'* mentioned before, as well as for function '*put_message*' which is never invoked from class '*AgentScapeApiImpl*'**.**

```
private MessageCenter mc;
public
void put_message (Envelope message, boolean canForward)
    throws MessageException
{
    mc.putMessage (message.copy(), canForward);
}

public
void put_messages (List<Envelope> messages, boolean canForward)
    throws MessageException
{
    List<Envelope> copies = new ArrayList<Envelope>(messages.size());
    for (Envelope env : messages)
        copies.add (env.copy());

    mc.putMessages (copies, canForward);
}
```

```
public
Envelope get_message (AgentID fromID,
        Collection<AgentHandle> handles, boolean block)
    throws AgentUnknownException, AgentIDUnknownException
{
    return mc.getMessage (fromID, handles, block);
}

public
List<Envelope> get_messages (AgentID fromID,
        Collection<AgentHandle> handles, boolean block)
    throws AgentUnknownException, AgentIDUnknownException
{
    return mc.getMessages (fromID, handles, block);
}
```

5.2.6 Class MessageCenter

This class is implementing functions '*putMessage*', '*putMessages*', '*getMessage*' and '*getMessages*'. Function '*putMessage*' invokes directly function '*putMessages*' passing a single element in the set of messages.

```
public void putMessage (Envelope env, boolean canForward)
        throws MessageException
{
    ArrayList<Envelope> envs = new ArrayList<Envelope>();
    envs.add (env);
    putMessages (envs, canForward);
}
```

Function '*putMessages*' delivers all messages provided. This is done locally if the receiver is on the same container, or invoking '*putMessages*' function of a message manager in a remote container.

```
public void putMessages (List<Envelope> envs, boolean canForward)
        throws MessageException
{
if (a.equals(localAddress))
{
    try
    {
        deliverLocally (list);
else if (canForward)
{
    HostManager mc = null;
    try
    {
        mc = lc.getMessageCenter (a);
        mc.put_messages(list, false);
    }
```

Both functions '*getMessage*' and '*getMessages*' invoke function '*dequeMessages*' which is already available on the message manager.

```
public List<Envelope> getMessages (AgentID fromId, Collection<AgentHandle> handles,
                                   boolean block)
    throws AgentIDUnknownException, AgentUnknownException
{
    return dequeueMessages (fromId, handles, block, true);
}
```

5.2.7 Local Monitor integration proposal

To integrate ACL module with the local monitor component, an observation must be submitted to the monitor each time a message is sent or received by an agent. This has to be done in a transparent

```
public Envelope getMessage (AgentID fromId, Collection<AgentHandle> handles,
                            boolean block)
    throws AgentIDUnknownException, AgentUnknownException
{
    List<Envelope> result = dequeueMessages (fromId, handles, block, false);

    if (result == null || result.size() == 0)
        return null;

    return result.get(0);
}
```

way from AgentScape programmer's point of view, that is, in order to integrate the ACL module with the local monitor, the programmer should not need to add extra code to the agent. This constraint discards the option of performing the integration in the class 'Agent'. Class 'AgentScapeApiImpl' is not either a good idea for performing the integration, because it is only one of the three implementations of interface 'AgentScapeApi'. However, class 'LocalHostManager' is a good starting point to perform the integration. In this class, a different 'MessageCenter' (i.e. 'MonitorizedMessageCenter') can be used to instantiate the variable 'mc'. This class can be just an extension of the original 'MessageCenter' where functions 'dequeMessages' and 'deliverLocally' are modified in order to submit observations to the local monitor when a message is received or sent respectively.

6 Global Monitor

[TBD]

7 Monitor Tool

[TBD]