Demo:A BDI model for component & service-based systems: Self-OSGi

Mauro Dragone

1 Introduction

This paper illustrates the use of Self-OSGi - a novel agent toolkit built on the Open Service Gateway initiative (OSGi) to ease the construction of adaptive component & service-based software systems.

Component & service orientation is a highly popular approach for building adaptive software solutions, for instance, as part of Robotic and Smart Environment applications. Component frameworks operate by posing clear boundaries (in terms of provided & required service interfaces) between components and by guiding the developers in re-using and assembling these components into applications.

More recently, the same frameworks are also provided with limited run-time flexibility through late binding and dynamic composition of the components' interfaces. However, they fail to offer an adequate support for implementing adaptive systems, in terms of common adaptation models. They also rely on the static definition of service attributes and priorities, and only provide limited support for context-sensitive service selection, on-demand instantiation of components and error recovery strategies. All of these limitations make it difficult to apply a consistent adaptation strategy throughout entire applications.

Self-OSGi addresses these issues by leveraging previously unexploited similarities between component & service technologies and the Belief, Desire, Intention (BDI) agent model. Compared to other Self-* software initiatives, Self-OSGi injects agent-based mechanisms into the fabric of a mainstream component framework, defacto transforming components into agents. Self-OSGi enables the definition of fine-grained system's adaptation policies and leverages well proven agent-based adaptation mechanisms to drive the dynamic instantiation and selection of components and services.

Mauro Dragone
CLARITY Centre for SensorWeb Technology, University College Dublin (UCD), Belfield, Dublin, Ireland, e-mail: mauro.dragone@ucd.ie

2 Self-OSGi

OSGi defines a lightweight component container and service brokering framework, which facilitates the dynamic installation and management of modular units of deployment, called *bundles*. Bundles collaborate by way of services specified through Java interfaces.

Developers can also associate lists of name/value attributes to services, and components can query, at run-time, the OSGi Service Registry for services that match given search criteriae expressed as LDAP filters. Furthermore, *Declarative Services (DS) for OSGi* offers a declarative model for managing multiple components within each bundle and also for automatically publishing, finding and binding their required/provided services, based on XML component definitions. This has the advantage that components can be implemented as plain Java objects, without reference to the OSGi framework.

However, DS only matches pre-defined filters with pre-defined services' attributes of already active components, but does not consider the automatic instantiation of new components, the context-sensitive selection of their services, or the automatic recovery from their failure.

Self-OSGi addresses these issues by translating the BDI agent model into general component & service concepts. In particular, the separation between the services' interface and the services' implementation is the basis for implementing both the declarative and the procedural components of BDI-like agents, and also for handling dynamic environments, by replicating their ability to search for alternative applicable plans when a goal is first posted or when a previously attempted plan has failed.

Further documentation on how to setup and launch an OSGi container, such as the one included in the Java IDE Eclipse, can be found at *http://www.osgi.org*. Developers can simply add adaptive capabilities to their applications by launching the Self-OSGi Core bundle together with their bundles, with minimum or no interventions to their existing code.

Figure 1 illustrates the use of Self-OSGi with an example robot application. Goals, describing the desires that the robot agent may possibly intend, are represented by the interfaces of the services that may be used to achieve them - *service goals* in Self-OSGi. In the example, the robot uses the service goal *GoalBeAtLocation*, with the method *(void) BeAtLocation(X, Y)* to express the goal of being at a given *(X,Y)* location, and the service goal *(Localization*, with the method *Location getLocation()*, to represent the goal of tracking its own location.

Plans, describing the means to achieve goals, are represented by the components - *component plans* - implementing (providing) them. A component plan may require a number of service goals in order to post sub-goals, to perform actions, and also to acquire the information it needs to achieve its objectives. In the example, the robot has two different component plans that can be used to localize the robot, respectively, *LaserSLAM*, and *CameraSLAM*.

The following is part of the XML files describing the *MoveTo* and the CameraSLAM component plans.

```
<scr:component ... factory="MoveTo" name="MoveTo">
   <implementation class= "MoveToImpl"/>
   <service>
        <provide interface="GoalBeAtLocation"/>
   </service>
   <reference cardinality="0..1" interface= "Localization"
                   policy="dynamic" />
</scr:component>

<scr:component ... factory="CameraSLAM" name="CameraSLAM">
   <implementation class= "CameraSLAM"/>
   <service>
        <provide interface=" Localization"/>
   </service>
   <reference cardinality="0..1" interface= "Video" name="Video"/>
   <property name="self.osgi.precondition.LDAP" value="(light>30)"/>
</scr:component>
```

The definition of MoveTo declares its requirement of localization information as *dynamic*, in order to allow OSGi to activate it even when the reference to the Localization service goal is not resolved, thus avoiding to having to commit to a specific localization mechanism.

Noticeably, the definition of CameraSLAM includes special property fields, *self.osgi.precondition...*, whose value may be used to characterise the context when the component plan is applicable. Self-OSGi can support different syntaxes to express both *pre-conditions* and *in-conditions*, i.e. conditions that must be valid before or while the component plan is used. In the example, LDAP is used to describe how

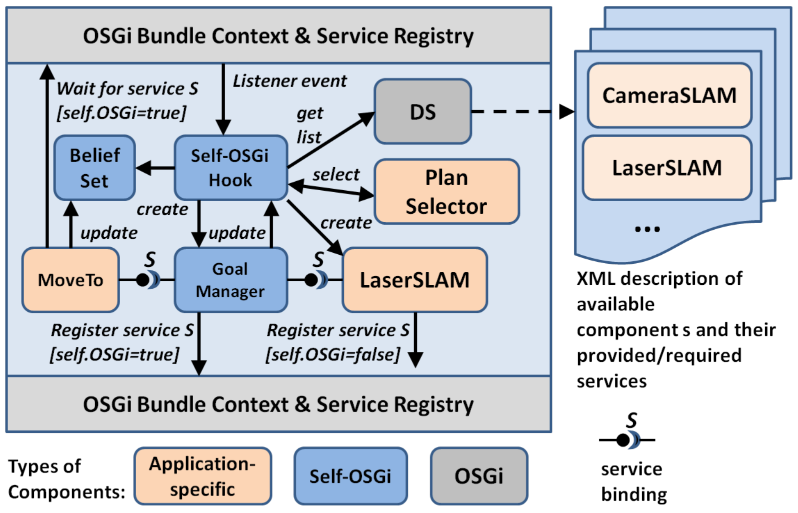
OSGi Bundle Context & Service Registry
Wait for service S [self.OSGi=true]
Listener event
DS
get list
Belief Set
Self-OSGi Hook
select
Plan Selector
create
update
update
create
MoveTo
S
Goal Manager
S
LaserSLAM
Register service S [self.OSGi=true]
Register service S [self.OSGi=false]
OSGi Bundle Context & Service Registry
CameraSLAM
LaserSLAM
...
XML description of available component s and their provided/required services
S
service binding
Types of Components:
Application-specific
Self-OSGi
OSGi

Fig. 1 Part of the Self-OSGi system discussed in the example

CameraSLAM can only be used when the intensity of the ambient light, e.g. sensed by a light sensor component, is believed to be above a given threshold.

Rather than using static service attributes, Self-OSGi tests conditions by matching them against the attributes stored in special *Belief Set* components, which can be updated, at run-time, by application component plans.

The following code requests the robot to go to a given location by initializing a standard OSGi *ServiceTracker* object to request the GoalBeAtLocation service goal. For simplicity, the code also sets the *light* belief (*beliefSet.add()*). In addition, it indirectly communicates with the Self-OSGi bundle to alter the standard Java call semantic by way of special parameters - starting with the prefix *self.osgi*. Specifically, it demands for the service goal to be re-attempted upon failure, at least as long as 120 seconds do not elapse from the first attempt. Other parameters may be used to define asynchronous and/or parallel execution threads among multiple service calls.

```
beliefSet.set(light, 55);
ServiceTracker tracker = new ServiceTracker(...,
   context.createFilter(
      "(&(objectClass="+GoalBeAtLocation.class.getName()+")" +
      "  (self.osgi=true)(self.osgi.Timeout=120))",)).open();

 (GoalBeAtLocation)(tracker.waitForService(0))
.beAtLocation(100, 200);
```

The tracker's request is intercepted by Self-OSGi, which queries the DS for the list of all the components able to provide the requested service (i.e. LaserSLAM and CameraSLAM in this case). After that, Self-OSGi implements the BDI cycle by (i) finding all the component plans with satisfied pre-conditions, and (ii) activating the most suitable one found by using user-provided, application-specific ranking components.

In order to define context and performance-aware adaptation policies, developers can write ranking components that access both the belief set and the statistics of past components' performances (e.g. call time, but also throughput and latency, in the case of service goal used to subscribe to periodic data updates). Performances are collected by installing a *Goal Manager* component (implemented using the Java *dynamic proxy* class) between the client that has originally requested a service goal, and the component activated to provide it. It is thanks to this mediation, that Self-OSGi can catch failures in services' activation and trigger the selection of alternative component plans.

In the robot example, these features are used to make the robot pursue its intended target while opportunistically exploiting any suitable localization component plan, for instance, starting with the CameraSLAM and then switching to the LaserSLAM if the first fails or if the ambient light drops below the given threshold.