COMPONENT & SERVICE-BASED AGENT SYSTEMS: SELF-OSGI

Mauro Dragone
CLARITY Centre for Sensor Web Technologies, University College Dublin (UCD), Belfield, Dublin, Ireland

Keywords: Autonomic software, Self-* software systems, Agent oriented software engineering, Component based software engineering.

Abstract: This paper proposes the adoption of the Belief- Desire-Intention (BDI) agent model for the construction of component & service-based software systems with self-configuring, self-healing, self-optimizing, and self protecting (self-*) properties. It examines component & service, and agent technologies, and shows how to build a component & service-based framework with agent-like autonomous features. This paper illustrates the design of one such framework, Self-OSGi, built over Java technology from the Open Service Gateway Initiative (OSGi). The use of the new framework is illustrated and tested with a simulated robotic application and with a dynamic service-selection example.

1 INTRODUCTION

Today, autonomic and adaptive software architectures are pursued in a number of research and application strands, including Robotics, cyber-physical systems, wireless sensor networks, and pervasive and ubiquitous computing.

In order to operate in these highly dynamic, unpredictable, distributed and open environments, these software systems must exhibit self-configuring, self-healing, self-optimizing, and self protecting (self-*) properties.

These problems are being addressed by both the Component-Based Software Engineering (CBSE) and the Agent-Oriented Software Engineering (AOSE) paradigms, each offering a modular design by which to encapsulate, integrate and organize the different systems functionalities

CBSE operates by posing clear boundaries between architectural modules (the *components*) and guiding the developers in re-using and assembling these components into applications. This typically involves an unambiguous description of the component's behavioral properties, and the set of their legitimate mutual relationships, in terms of provided and required interfaces (the *services*).

More recently, in order to adapt to varying resource availability and to increase system fault-tolerance, component frameworks are also provided with limited run-time flexibility through late-binding and dynamic wiring of component's interfaces.

Noticeably, this makes component & service based systems already resemble AOSE solutions, by favouring a component-centric rather than a global system perspective. However, they fail to provide an adequate support for these adaptive implementations, in terms of a common adaptation model and/or adaptation steps. Thus, the developer has to write custom, application-specific adaptation code. This makes difficult to ensure that a consistent and interoperable adaptation strategy is applied throughout all the components in one application, and also to maintain, and re-use these strategies across multiple applications.

In contrast, AOSE provides a method of abstraction and system decomposition based on *agentification*. This transforms a software application into a goal-oriented, autonomous agent by building a wrapper around it so it can interoperate with the rest of the system through standard, Agent Communication Language (ACL) interfaces and associated coordination protocols.

However, the emphasis of AOSE and associated multiagent programming platforms and toolkits is the coordination of large scale, deliberative multiagent systems (MASs) while issues arising from low-level, application specific functionalities are often overlooked. Consequently, as noted in (Kim, 2005), using an agent platform limit the acceptance of mobile agents as simple programming constructs, as the programmer is forced to center its development, its programming units, and its whole applications on the concept of agent. Rather than a middleware-type complement to traditional (object/component-oriented) software development, agents then become

Dragone M..
COMPONENT & SERVICE-BASED AGENT SYSTEMS: SELF-OSGI.
DOI: 10.5220/0003890302000210
In *Proceedings of the 4th International Conference on Agents and Artificial Intelligence* (ICAART-2012), pages 200-210
ISBN: 978-989-8425-95-9

the frontware and require the definition of complex interfaces toward the application and operating system resources.

These problems are only partially addressed by recent attempts to standardise these interfaces, such as with the SoSAA framework (M. Dragone and O'Hare, 2009b), and the Environment Interface Standard (T. M. Behrens and Hindriks, 2009), While all these approaches promote the interoperability in the way agents can operate in different environments, in different measures they also worsen the barrier between agent-based solutions and other mainstream software engineering approaches by advocating a clear-cut separation between agent - providing the intelligence - and non-agent approaches - carrying out application-specific functions.

These are the main motivations for seeking tightly integrated architectures that leverage the different characteristics and advantages of AOSE and CBSE. In particular, the focus of this work is the unification between agent, component and service concepts in a single methodology for the construction of autonomic software systems with Self-* properties. On one hand, this work aims to define a set of re-usable, modular end extensible adaptation mechanisms for component & service-based systems. On the other hand, the same approach will produce modular and lightweight agent systems that are tightly integrated with mainstream component & service technology.

The remainder of the paper is organized in the following manner: In order to draw a parallel between the CBSE and AOSE paradigms and guide the design of hybrid CBSE/AOSE systems, Section 2 summarises the popular Belief-Desire-Intention (BDI) agent model while Section 3 examines the most important characteristics of component & service frameworks. Section 4 provides an overview of the most significant agent/component integration approaches attempted in past research. Section 5 translates the BDI agent model into component & service-based concepts. Section 6 introduces Self-OSGi, a novel agent toolkit, which is developed using the Open Service Gateway initiative (OSGi) component & service technology (OSGi, 2011). Section 7 illustrates the use of Self-OSGi and tests its performance with a robotic application and a dynamic service-selection example. Finally, Section 8 summarizes the contributions of this paper and points to some of the directions to be explored in future research.

2 THE BDI AGENT MODEL

The Belief, Desire, Intention (BDI) is undoubtedly the most popular agent model, with many implementations directly related to Rao & Georgeff's abstract BDI architecture (Rao and Georgeff, 1992) and its Procedural Reasoning System (PRS) implementation (Georgeff and Lansky, 1985).

Kinny et al. (D. Kinny and Rao, 1996), describes the design of a BDI agent in terms of three components:

- A Belief Model, describing the information about the environment and internal state that an agent may hold, together with the actions it may perform.
- A Goal Model, describing the desires that an agent may possibly intend, and the events to which it can respond
- A Plan Model, describing the set of plans available to the agent for the achievement of its goals

While these traits were introduced to allow the computational tractability of the model, they are now recognized as being the most distinctive characteristics of BDI systems in general for their ability to support rational, resource-bounded reasoning in dynamic and uncertain domains.

Beliefs, for example, are essential since an agent has limited sensory ability and needs to build up its knowledge of the world over time. In this sense, beliefs - usually represented with first order logic predicates - serve as a cache with which the agent can remember past events or other important information that could be costly to re-compute from raw perceptual data or inferred logically.

The distinctions between goals and plans constitute an important source of modularity that contributes to the agent's ability to cope with contingencies. The fundamental observation is that goals, as compared to plans, are more stable in any application domain and multiple plans can be used/attempted to achieve the same goals. This also allows examining the application domain in terms of what needs to be achieved, rather than the types of behaviour that will lead to achieving it.

PRS implements a computationally tractable BDI model with the following simplifying assumptions:

- The system explicitly represents beliefs about the current state of the world as a ground set of literals with no disjunctions or implications (as in STRIPS).
- The system represents the information about the means of achieving certain future world states and the options available to the agent as pre-compiled plans.

Each plan in the plan library can be described in the form $e : \Psi \leftarrow P$ where P is the body of the plan,

e is an event that triggers the plan (the plan's post-conditions), Ψ is the context for which the plan can be applied (which corresponds to the preconditions of the plan).

The body of each plan is a procedural description containing a particular sequence of actions and tests that may be performed to achieve the plan's post-condition. Plans may also post new goal events, leading AND/OR goal-plan execution trees. For a plan to succeed all the subgoals and actions of the plan must be successful (AND); for a subgoal to succeed one of the plans to achieve it must succeed (OR).

When a plan step (an action or sub-goal) fails for some reason, this causes the plan to fail, and an alternative applicable plan for its parent goal is tried. If there is no alternative applicable plan, the parent goal fails, cascading the failure and search for alternative plans one level up the goal-plan tree.

Goal events in PRS are posted by using special temporal operators like *achieve* and *maintain*. The achieve operator is used to request the achievement of a new goal. For instance, an agent in control of a thermostat may be instructed with a new set-point temperature by posting the goal *achieve(T>20)*. The *mantain* operators specify a homeostatic goal - one that must be re-achieved if it ever becomes unsatisfied.

The ability to search for alternative applicable plans when a goal is first posted or when a previously attempted plan has failed enables these systems to handle dynamic environments. The final decision of which plan to activate is performed using meta-level procedures implementing application-specific strategies, for example, by considering user-defined priorities.

Finally, in PRS-like BDI systems, desires and goals are represented only in the transient form of goal events (posted by the application), while the intentions to purse them is stored implicitly in the stack of plans they triggered. This poses an obstacle to the effective decoupling between plans and goals, forcing, for instance, the agent to drop goals for which no feasible plan can be attempted at the time the goal is posted. Such an issue is addressed in modern agent systems, such as Jadex, by incorporating explicit and declarative goal representations into the agent interpreter in order to ease the definition of goal deliberation strategies (Alexander Pokahr, 2005).

3 COMPONENTS & SERVICES

The Open Service Gateway Initiative (OSGi), CORBA Component Model, Microsoft Object Model, Enterprise JavaBeans, and Fractal are some of the mainstream component-enabling technologies used for the creation of many industrial-strength software systems. Conceptually, the same technologies also provide a composite model for service oriented architectures, by helping to design systems in terms of application components that can expose their public functionality as services as well as invoke services from other components.

3.1 Component Containers

One of the most important common concepts among component-enabling technologies is the relationship between a component and its environment, wherein a newly instantiated component is provided with a reference to its container or component context. The component container can be thought of as a wrapper that deals with technical concerns such as synchronisation, persistence, transactions, security and load balancing. The component must provide a technical interface so that all components will have a uniform interface to access the infrastructure services. For instance, a common solution to implement activity-type components, i.e. components that need to attend to their process rather than merely react to events, is to segment these activities in steps, which are then executed by a scheduler - usually shared among multiple components.

Most relevant for this paper, OSGi defines a standardised component model and a lightweight container framework, built above the JVM. OSGi is used as a shared platform for network-provisioned components and services specified through Java interfaces. Each OSGi platform facilitates the dynamic installation and management of units of deployment, called bundles, by acting as a host environment whereby various applications can be executed and managed in a secured and modularised environment. An OSGi bundle organises the frameworks internal state and manages its core functionalities. These include both container and life cycle operations to install, start, stop and remove components as well as checking dependencies.

3.2 Component & Service Adaptation

Many of the infrastructural services associated with component contexts act as late-binding mechanisms that can be used to defer inter-component associations by locating suitable collaboration partners. Through these brokering mechanisms, components do not need to be statically bound at design/compilation time but can be bound either at composition-time or at run-

time in order to favour the construction of adaptable software architectures.

For instance, the *Activator* class in OSGi, the *BeanContext*, and the components membrane in Fractal enable components to look up services in the frameworks service registry, register services, access other components, and install additional components within the local platform.

The separation between component's services and their actual implementation is the key to the creation of self-managing and adaptable architecture.

In striving toward these solutions, a formal base is usually required to describe the provided and required features of individual components and also important semantic aspects, such as the correct way those features are to be used. With OSGi, developers can associate lists of name/value attributes to each service, and use the LDAP filter syntax for searching the services that match given search criteria. Furthermore, Declarative Services (Humberto Cervantes, 2003) 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. This minimizes the amount of code a programmer has to write; it also allows service components to be loaded only when they are needed (Delayed Activation). Declarative Services indicates if a required service is mandatory or optional. The binding makes the life cycle of the component dependent on the presence of that linkage, respectively having its state as active or passive depending on the presence or absence of the component's dependencies.

These mechanisms are not limited to components and services running on a single platform. Remote service bindings are usually achieved through port and proxy mechanisms. For example, in both R-OSGi and D-OSGi, remote bindings can be viewed as connection points on the surface of the component where the framework can attach (connect) references to provides-ports provided by other components. The framework is then responsible for returning the correct Java object when a port is requested by a component. It either calls the appropriate methods of the locally available service implementation object or translates the Java method calls to messages, sends them to a remote container (e.g., availing of Java RMI, SOAP, or JXTA), waits for remote execution and then returns the value contained in the received message.

In the OSGi implementation OSCAR, the same mechanism is also used to support intelligent hot swapping of services to implement fault-tolerant systems. Specifically, as every service in OSGi may be given a certain rank which can be used to describe its quality and importance, when queried about a particular service, OSCAR automatically tries to locate the highest-ranked implementation.

A-OSGi (J. Ferreira and Rodrigues, 2009) goes a step further by providing a number of mechanisms that can be used to create self-adaptive architectures. Firstly, a monitor component measures the CPU and memory used by each bundle by: (i) altering the OSGi life cycle layer so that all the threads in a bundle belong to the same thread group, and by (ii) providing each service client with a proxy that executes the service methods within the same thread group. Secondly, a planning component interprets Event Condition Action (ECA) rules specifying adaptation actions to be executed in response to specific events and given conditions. Finally, an execution component applies these actions to their target components. Specifically, A-OSGi considers three main action types, namely: 1) specify rules for service bindings, in such a way that a specific bundle is prohibited, or obliged, to use some specific service implementation; 2) change service properties, for instance change a parameter associated with a service implementation; and 3) control the life cycle of a bundle, by either starting or stopping bundles.

4 RELATED WORK

A number of works have looked at leveraging both component and agent approaches for the development of adaptive software systems.

A component-based approach in the construction of multi agent systems has been supported by numerous researchers in the past. This typically considers the components to be simply the building blocks from which agents are constructed (M. Amor and Troya, 2003). An advantage of this approach is the ability to take domain-specific issues into account at the component level. The decisions made on these issues can therefore be separated from the task of constructing the multi agent system as a whole, thus simplifying the process. The resulting agent applications inherit some of the (functional/non-functional) properties from the underlying component framework. However, this form of technical integration does not contribute much to a conceptual combination of both paradigms as, once they are built, agents remain the only primary entity form.

A different integration approach is advocated in SoSAA (M. Dragone and O'Hare, 2009b), in which an high-level agent framework supervises a low-level component-based framework. The latter provides a computational environment to the first, which then

augments its capabilities with its multi-agent organisation, ACL communication, and goal-oriented, BDI-style reasoning. A SoSAA Adapter interface provides meta-level sensors and meta-level actuators to operate on the component layer, to load, unload, configure components, observe their internal status, and bind their provided/required interfaces. Components are left to automatically carry out lower-level behaviours and can interact through a variety of non-ACL collaboration styles, including method calls, messages and events. The deliberative layer makes decisions about when such behaviours and communication mechanisms are necessary or desirable in order to satisfy overall system and application requirements. However, keeping neatly separated components and agents fails to contribute much in consolidating both paradigms. Furthermore, the use of two separate frameworks means that the resulting systems are subjected to both development and run-time overheads.

Removing the need for a separate infrastructure shared by a large number of distributed applications is what motivates the approach followed in the M&M framework (Kim, 2005). In contrast to application development centred upon agent platforms, M&M applications become agent-enabled by incorporating well-defined binary software components into their code. These components give the applications the capability of sending, receiving and interacting with mobile agents. The applications themselves are developed using the current industry best practice software methods (*JavaBeans*) and become agent-enabled by integrating the mobility components. Such an approach succeeds in moving some agent mechanisms into the middleware layer. However, M&M only addresses agent mobility while components are not equipped with goal-oriented reasoning capabilities.

More recently, the *Active Component* (AC) concept (Lars Braubach, 2010) has been proposed as a way to integrate successful concepts from agents and components as well as active objects and make those available under a common umbrella. Active components are autonomous acting entities (like agents) that can use message passing as well as method calls (like active objects) for interaction. They may be hierarchically structured and are managed by an infrastructure that ensures important non-functional properties (like components).

The AC framework has been realized in the Jadex AC platform. In particular, Jadex AC runs on an extended Jade platform and supports component types (kernels) for BDI agents, as well as simpler, task-specific agent models. Noticeably, such an approach defines a proprietary component or agent frameworks and does not leverage mainstream component-based initiatives and standards, such as OSGi.

5 COMPONENTS & SERVICES AGENT MODEL

In order to inform the design of component & service agent systems, and before dwelling on the details of Self-OSGi's implementation, this section translates the BDI model into general component & service concepts.

5.1 Modular Belief Model

Rather than storing all the agent's beliefs into a single, centralized belief set, a component & service-based organization can be used to access and distribute the processing of information across the system. Specifically, an agent may use a number of sensor components to interface with its environment. Each of these components produces data that can be exported with any of the collaboration styles afforded by mainstream component & service-enabling technologies (procedural calls, messaging, events). This information can be fed to other perception components, for instance, to infer situations or test conditions involving multiple beliefs.

Such an approach enables the construction of a variety of perception architectures to fit with the sensors available to the agent and also with the run-time requirements posed by the specific agent's perceptual processes. In addition, since each sensor and perceptual process must provide a strongly typed definition of their required/provided services, such clear dependencies pose an upper-bound to the amount of data that must be processed at any given time by each component.

5.2 Service Goal Model

Similarly to the separation the separation between component's services and their actual implementation, the separation between goals and plans in BDI agents is the key to the creation of self-managing and adaptable architecture.

To this end, it is useful to introduce the concept of ***Service Goal***, informally defined as *the interface of a service that may be used to achieve one of the agent's goals*.

Service goals may represent either: (i) sub-goals defining the desired conditions to bring about in the world and/or in the system's state (for instance, the service *(void) atLocation(X, Y)* may be used by a

robotic agent to represent the goal of being at a given location), and (ii) sub-goals subtending the exchange of information. Service goals that are used to access data and to subscribe to data updates and event notifications fall into the latter category. For instance, the service goal *Image getImageCamera()* may be used by a robotic agent to express the goal of retrieving the last frame captured by one of its cameras.

In addition, service goals' attributes may be used to further characterise each service goal, e.g. the characteristic of the information requested/granted, as well as important non-functional parameters. For instance the *atLocation* service goal may have the attribute *Min/MaxVelocity* to specify the minum/maximum velocity the robot should/may travel. The attribute *MinimumFrameRate* may be used to specify the mimum frame rate for the image captured with the *getImageCamera()* service, while the *Side* attribute, with values in {*left, right*}, may be used to specify which one of the robotic cameras must be used.

5.3 Component Plan Model

A ***Component Plan*** is informally defined as *a component implementing (providing) a service goal (its post-condition)*. 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 post-condition. Component plans may attend their activities with their own thread of control. In addition, they may react to incoming messages/events, and also export functions to a scheduler used for control injection. For instance, a *MoveTo* component plan may process the images from a robot's cameras and control the velocity and the direction of the robot to drive it safely toward a given location. The same component plan may subscribe to impact alert notifications generated by the onboard bumper sensor, and stop the robot upon the reception of one such alert.

5.4 Goal Manager

Section 2 has discussed how an agent must rely on explicit representations of its own goals in order to keep track of goals achieved and yet to achieve.

To this end, it is useful to introduce the concept of ***Goal Manager***, that is, a component used to decouples the plan requesting a service goal from the component plan ultimately providing it. Invoking a service goal should first trigger the activation of the corresponding Goal Manager, which then will take care to invoke one of the component plans able to achieve it.

Thanks to its mediation, a Goal Manager can be used to re-invoke the same service goal upon failure of the component plan first used to achieve it. Crucially, further invocations may use different implementations of the service, i.e. different component plan options. In addition, the Goal Manager can be used to maintain execution statistics for each component plan option, in order to drive the future selection of the best suitable one, e.g. the one less likely to fail, and/or with better performance.

6 SELF-OSGI IMPLEMENTATION

Figure 1 shows a diagram of the main classes involved in the operation of Self-OSGi, including pre-existing OSGi classes, and a couple of example application component plans (*X* and *Y*)

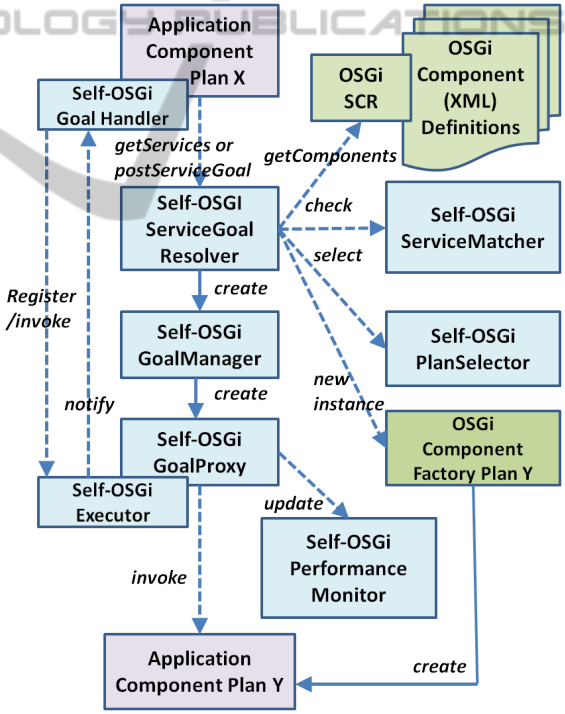

Figure 1: Self-OSGi System Diagram.

The ***OSGi Service Component Runtime (SCR)*** is part of the Declarative Service (DS) bundle. Components are activated and deactivated under the full control of SCR. SCR bases its decisions on the information in the components definition stored in XML files, which describe the services that are implemented by the component, and its dependencies on other services. SCR will automatically register every service

provided by the component into the central service registry of the platform, and bind every service required by the components with those available in the registry.

If a components description specifies the *factory* attribute of the component element in its XML definition, SCR will register a ***OSGi Component Factory*** service. This service allows client bundles to create (*newInstance()*) and activate multiple component, on demand, and dispose of them after they have been used.

The ***Self-OSGi ServiceGoalResolver*** provides a transparent resolution service (*postServiceGoal()*), which can be used explicitly by application component plans to get the reference to the implementation of the service goals they need. In the example in the diagram, the ServiceGoalResolver is used to get a reference to an implementation of the service goal *G1* to be called by the component plan *X*. Alternatively, the ServiceGoalResolver can register the ***OSGi Framework Find Hook Service*** in order to be called during framework service find (get service references) operations. Noticeably, the latter approach does not require any intervention to the application code, (which can be developed in terms of plain old java objects (POJOs), thus making very easy to apply Self-OSGi to pre-existing applications.

On its own, the OSGi SCR is only able to satisfy a service request by looking at components already active or pending activation. However, the agent execution model requires the on-demand activation of component plans. For this reason, the ServiceGoalResolver collaborates with the ***OSGI Factory Component Service*** (not depicted in the diagram) and with the SCR to create and bind component plans on demand.

Once called by the framework or called explicitly by component plans in need for services, the ServiceGoalResolver creates a ***Self-OSGi Goal Manager***, which will install a ***GoalProxy*** object (implemented using the Java *dynamic proxy* class) by registering it to the OSGi service registry in place of the original requested service.

After that, the SCR will automatically bind the GoalProxy to the component plan that has requested the service goal. However, only when the service goal is finally invoked, the GoalProxy will trigger the selection of a suitable service goal's implementation. To do this, the GoalProxy will contact the GoalManager, which will then ask the ServiceGoalResolver to trigger the selection of the most suitable component plan implementing the requested service goal.

The ServiceGoalResolver will use the ***Self-OSGi ServiceMatcher*** and the ***Self-OSGi ComponentPlanSelector*** to identify, respectively, (i) which components are suitable to be executed in the current context (options), and (ii) which option, among all the suitable ones, is deemed to be the best to satisfy the request at hand.

By default, the ServiceMatcher filters out component plans by using the LDAP filter query specified with the original service request (*getServices*). However, in order to more accurately replicate the BDI agent model, it is important to be able to check the feasibility of a particular component plan at the time its post-condition service goal is invoked, also by examining the value of the parameters used in its invocation.

To this end, Self-OSGi relies on a programming convention, which demands the component plan to postpone the actual execution of the service goal (and also the loading of service sub-goals) until it verifies that the service goal can be attempted in the current context. It is the responsibility of the component plan's developer to (i) make sure that the verification of the precondition will be as lightweight as possible, and (ii) that it will raise an exception if the component plan should not be used in the current context. In this manner, Self-OSGi will quickly skip all the unsuitable options (in the order they are ranked by the ComponentPlanSelector) before executing the service goal.

The Goal Proxy measures execution statistics (CPU system and user time) for both synchronous and asynchronous invocation to its corresponding service goal. A number of dedicated attributes in the original service goal request can be used to alter the call semantic, including *MaxAttempts* (used to specify the number of invocation to be attempted upon failure), and *DelayBetweenAttempts* (used to specify a delay between each attempts). Crucially, after each failed attempt, the goal proxy will contact the Goal Manager, in order to trigger the resolution of a (possibly) different implementation (plan option).

A ***Self-OSGI GoalHandler*** class may be used to support asynchronous operations. If the requester registers a GoalHandler, the goal proxy will schedule each invocation with a platform's scheduler (implemented over the *java.util.concurrent.ExecutorService* class). Group-type specializations of the goal handler class exist, such as *ANDGoalHandler* and *ORGoalHandler*, which are used, respectively, to define conjunctions and disjunctions of groups of service goals. Noticeably, using group goal handlers breaks the POJO programming model, as the developer needs to incorporate these Self-OSGi specific classes into its code.

Finally, the ***Self-OSGI PerformanceMonitor***

class collects the performance statistics for all the component plan options. This class associates a priority to each component option. By default, the priority is computed by considering both the success rate (the rate between the number of successful invocation and the total number of invocation) and the speed to which the component had satisfied past requests.

7 TESTS

Self-OSGi is being used to re-factor a number of pre-existing applications, including robot control and distributed information retrieval systems. Developers can add Self-* capabilities to their OSGi applications by using the Self-OSGi Core bundle (which consumes 31K on the file system).

By way of example, the following code is part of a robot navigation test that instructs a mobile robot to move towards a given location while trying to detect a soccer ball via its on-board camera. The example reported here does so by explicitly using the Self-OSGi *postServiceRequest()* function to resolve two service goals, respectively *RecognizeObject* and *BeAtLocation*, before adding them to the same ANDGoalHandler and waiting for the successful completion of both service goals.

```
// Navigation \& Object Tracking Test
g1 = (ObjectRecognition)
     serviceResolution.postServiceRequest
     (ObjectRecognition.class.getName(),
     "&(maxAttempt=10)(object=ball)");
// set property for reliable navigation service
...
g2 = (BeAtLocation)
    serviceResolution.postServiceRequest
    (BeAtLocation.class.getName(),
     "&(maxAttempt=10)");
g1.subscribeObjectRecognitionEvents();
g2.beAtLocation(x,y);
andGoalHandler.add(g1).add(g2).Wait();
```

The following is part of the XML definition used to describe a *NavigateViaLaser* component plan able to achieve the beAtLocation service goal (as specified in the emphservice XML element).

```
<scr:component xmlns:scr=
    "http://www.osgi.org/xmlns/scr/v1.1.0"
    factory="NavigateViaLaserFactory"
    name="NavigateViaLaser">
   <implementation class=
      "ucd.robotics.impl.NavigateViaLaserImpl"/>
   <service>
       <provide interface=
       "ucd.robotics.goals.GoalBeAtLocation"/>
   </service>
   <reference cardinality="0..1" interface=
        "ucd.osgi.robotics.sensors.Laser"
         name="Laser" policy="dynamic"/>
   <reference cardinality="1" interface=
        "ucd.osgi.robotics.sensors.Battery"
         name="Battery" policy="static"/>
   <reference cardinality="1" interface=
        "ucd.osgi.robotics.actuator.Driver"
         name="Driver" policy="static" />
   <reference cardinality="1" interface=
        "ucd.osgi.robotics.data.Location"
         name="Location" policy="static" />
</scr:component>
```

The component requires (see the *reference* XML elements) the ability to: (i) receive laser data, in order to avoid obstacles along its path, (ii) receive position updates which report the current location of the robot, (iii) send control velocities to drive the robot's platform, and (iv) check the energy level of the robot's batteries (as its pre-condition), in order to make sure that the laser is activated only when the battery level is deemed sufficient to last until the robot achieves the target destination. Laser data, sonar data, location, robot's driving system, and battery's energy data can be accessed, respectively, via the *Laser*, *Sonar*, *Localization*, *Driver*, and *Battery* components, as shown in Figure 2. Of these components, only the Laser and the Sonar components must be initialized on-demand, while all of the other components are always active. As such, the XML definition of the NavigationViaLaser component plan specifies that the component plan can be instantiated even if the reference to the Laser is not resolved (the Laser reference is declared *dynamic* in the XML). This enables Self-OSGi to instantiate the NavigationViaLaser component without loading the laser driver, in order to check if there is sufficient energy for it to work should it be activated.

The following is the first part of the implementation of the BeAtLocation() service goal in the NavigateViaLaser component. The implementation loads the battery service goal and raises an exception if the remaining energy is not sufficient to complete the requested service goal, before subscribing to the laser data (and thus triggering the activation of the laser driver) and start controlling the robot.

```
class NavigateWithLaserImpl
   implements BeAtLocation {
  ...
void beAtLocation(int x, int y) {
   ..
energyNeeded = estimateEnergy(
   location.getX(), location.getY(),
   x, y);

if (battery.getEnergyLeft()<energyNeeded)
   throw Exception("Not enough energy!");

// load the laser
```

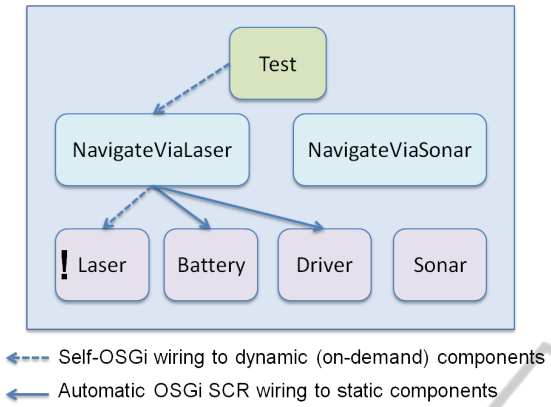

Figure 2: Part of the architecture of the robot system used for the tests described in the text. The figure depicts the wiring of the system before the failure of the Laser component. After the failure, Self-OSGi kept carrying out the test by using the NavigateViaSonar component after wiring it with the Sonar and the Driver component.

```
g1 = (Laser)
    serviceResolution.postServiceRequest
    (Laser.class.getName(), null);
g1.subscribeLaserData();

// drive the robot
...
```

Based on these instructions, Self-OSGi will pursue both the RecognizeObject and the BeAtLocation service goals in parallel and take care to instantiate the component plans that are most appropriate to the current situatios. For instance, Self-OSGi will consider the NavigateViaLaser component plan only when the robot has sufficient energy left to bring the robot to its intended destination. In addition, in the case that the battery level will still fall below the threshold and the laser stops functioning while NavigateViaLaser is active, Self-OSGi will automatically replace the NavigateViaLaser with the NavigateViaSonar component plan.

The real robot system does not really benefit from the speed of the dynamic service selection and replacement performed by Self-OSGi, as the robot needs time to initialize both the laser driver and the sonar driver.

Figure 3 shows the result of a fault-tolerance test run performed on a simulated version of the robot system depicted in Figure 2, where simulated laser and sonar components had no initialization latency, and with a simulated robot driving system with infinite accelleration. The testing environment consisted of a Pentium dual core at 2.40 GHz with 8GB of SDRAM, with the Sun J2SE 6.0 platform compliant JVM and running the Linux 2.6.24 kernel. The test simulated a failure in the Laser component after the robot's battery level dropped below the minimum level necessary to support the correct functioning of both the laser and the robot's driving platform. As soon as the NavigateWithLaser component plan failed to receive laser updates, it raised an exception that was captured by the Self-OSGi GoalProxy, which then triggered the selection of an alternative component plan to complete the GoalBeAtLocation goal. This resulted in the activation of a NavigateViaSonar component plan (driving the robot at lower velocity), and the subsequent activation of the Sonar component. Over 10 runs, the system was able to recover from failure with an average of 72 ms. For comparison, a similar dynamic service replacement performed over the Java Beans component framework and co-ordinated by a standard BDI agent toolkit took an average of 312ms (M. Dragone and O'Hare, 2009a).

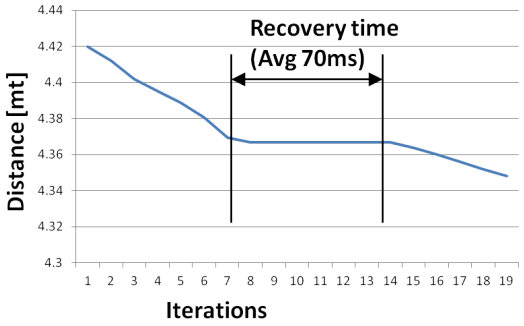

Figure 3: An extract from the results of the fault-tolerance test described in the text. X Axis - number of control iterations (in average one every 10 [ms]). Y Axis - distance from target destination [mt]. The robot stopped at iteration #7 when the the laser stopped working for the time Self-OSGi took to replace the laser with the sonar-based MoveTo behaviour.

Finally, Figure 5 shows the execution times obtained in experiments performed executing the root goal of the goal/plan tree depicted in Figure 4. Three possible component plan options existed to achieve the root goal while each of those options required the execution of two services, each with three possible implementations. In order to demonstrate the Self-OSGi's ability to measure and take into account the execution time of service goals, each component plan was programmed to require a different CPU time while the Self-OSGi PlanSelector component was programmed to assign greater priorities to previously unexplored plan options. The figure demonstrates how, when Self-OSGi was repeatedly asked to achieve the root goal, it automatically tried new component plan implementations at each iteration, ultimately converging on the best policy to achieve the root goal in the shortest time.

With the same setup, the overhead imposed by the

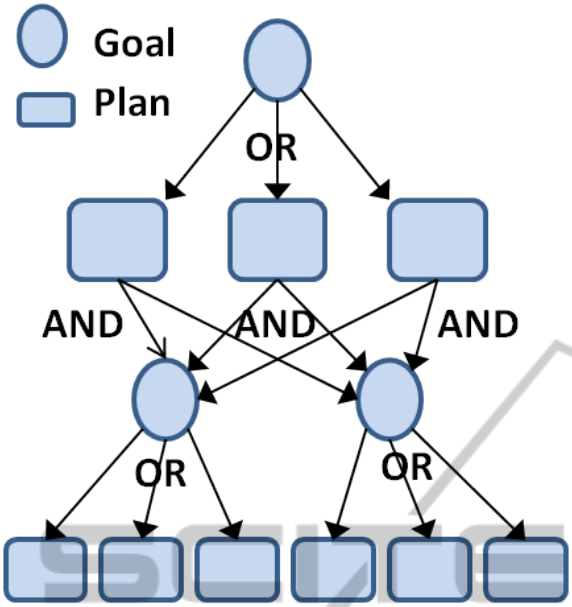

Figure 4: Structure of the test program.

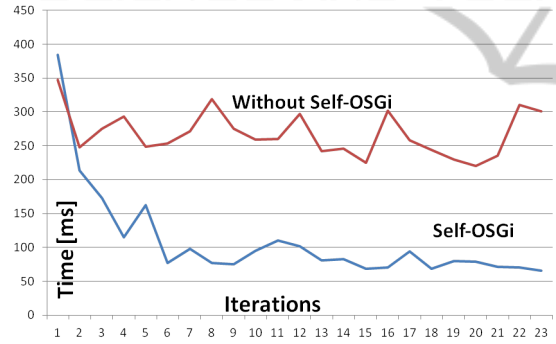

Figure 5: Service selection performance.

Self-OSGi service management (performance measurement and proxy mechanisms) was measured by comparing the time needed to achieve the root service goal with and without the installation of a GoalProxy. The average time added by the service management to each service goal was 0.15 milliseconds.

8 CONCLUSIONS AND FUTURE WORK

This paper has examined component, service and agent concepts, and has illustrated the design and the implementation of the Self-OSGi framework for the construction of systems with Self-* properties. Self-OSGi is built over OSGi technology by leveraging previously unexploited similarities between component & service and the BDI agent model.

Compared to similar CBSE initatives, such as the A-OSGi framework reviewed in Section 3.2, Self-OSGi provides re-usable, lightweight, modular end extensible adaptation mechanisms at component-level granularity that are also tightly integrated with the OSGi Declarative Service framework. Self-OSGi can be used to drive the selection of services, control the on-demand instantiation of the components implementing them, and monitor their performance to drive their future selection and to recover from failure. In contrast, A-OSGi can be used to control and monitor entire bundles, but does not offer any mechanism to discern the performance among the single components and services inside the bundle or to instantiate them on-demand.

In addition, the association with the BDI model allows Self-OSGi to leverage well-defined adaptation policies and results from BDI-related research.

Compared to existing AOSE/CBSE integration approaches, such as the SoSAA and the AC framework reviewed in Section 4, Self-OSGi provides an highly modular realization of the BDI agent model, which is grounded in the mechanisms offered by a mainstream component & service technology. This results in low performance and footprint overheads and fast system's adaptation, as shown in Section 7. Noticeably, existing agent platforms, such as JADE, have already been made compatible with the OSGi framework. However, this is usually done by encapsulating the entire agent platform into a single, monolithic OSGi bundle. Such an approach does not benefit of the increased modularity enabled by the OSGi framework.

In contrast, one of the goal of the Self-OSGi framework is to evolve into a modular agent platform. To this end, the simulated robot system and the service-selection benchmark test will be released as open source. An ACL bundle will also be realesed to provide FIPA-compliant ACL interoperability with existing agent toolkits.

Future work with Self-OSGi will seek to adapt agent/planning integration and agent learning techniques to tackle some of the main limitations of adaptive component & service frameworks, such as their lack of look-ahead and logical inference capabilities and their reliance on hard-coded pre-conditions of component plans.

The other direction to this work is the evaluation of the usability of Self-OSGi in a range of application domains in need of Self-* sofware architectures, including the control and monitoring of Wireless Sensor Network (WSNs), Ambient Assisted Living (AAL), and home automation systems.

ACKNOWLEDGEMENTS

This work has been partially supported by the EU FP7 RUBICON project (contract n. 269914) and by Science Foundation Ireland (SFI) under grant 07/CE/I1147.

REFERENCES

Alexander Pokahr, Lars Braubach, W. L. (2005). A goal deliberation strategy for bdi agent systems. In *MATES-2005*. Springer-Verlag.

D. Kinny, M. G. and Rao, A. (1996). A methodology and modeling technique for systems of bdi agents. In *Proc. Of 7th European Workshop on Modelling Autonomous Agents in Multi-Agent Worlds, LNAI1038*. Springer-Verlag.

Georgeff, M. and Lansky, A. (1985). *A system for reasoning in dynamic domains: Fault diagnosis on the space shuttle*. Technical Note 375, Artificial Intelligence Center, SRI Internationa.

Humberto Cervantes, R. S. H. (2003). Automating service dependency management in a service-oriented component model. In *In Proceedings of the Sixth Component-Based Software Engineering Workshop*.

J. Ferreira, J. L. and Rodrigues, L. (2009). *A-osgi: A framework to support the construction of autonomic osgi-based applications*. Technical Report RT/33/2009, May 2009.

Kim, H.-K. (2005). A component-based approach for integrating mobile agents into the existing web infrastructure. In *Third ACIS International Conference on Software Engineering Research, Management and Applications*.

Lars Braubach, A. P. (2010). Addressing challenges of distributed systems using active components. In *In Proceedings of 4th International Symposium on Intelligent Distributed Computing*.

M. Amor, L. F. and Troya, J. (2003). Putting together web services and compositional software agents. In *ICWE 2003, LNCS 2722*.

M. Dragone, R. Collier, D. L. and O'Hare, G. (2009a). Practical development of hybrid intelligent agent systems with sosaa. In *In Proceedings of the 20th Irish Conference on Artificial Intelligence and Cognitive Science (AICS 2009) Dublin Ireland*.

M. Dragone, D. Lillis, R. C. and O'Hare, G. (2009b). Practical development of hybrid intelligent agent systems with sosaa. In *Proceedings of the 20th Irish Conference on Artificial Intelligence and Cognitive Science, Dublin, Ireland*.

OSGi (2011). *Open Service Gateway Initiative (OSGi) http://www.osgi.org/Main/HomePage [Accessed October 2011]*.

Rao, A. and Georgeff, M. (1992). An abstract architecture for rational agents. In *In Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference KR*.

T. M. Behrens, J. D. and Hindriks, K. V. (2009). Towards an environment interface standard for agent-oriented programming (a pro-posal for an interface implementation). In *Technical report, Clausthal University*.