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

Mauro Dragone

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-* 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 on Java technology from the Open Service Gateway Initiative (OSGi). The use of the new framework is tested in a new test-bed designed to assess its ability to support Self-* software architectures.

1 Introduction

Today, autonomic and adaptive software architectures are pursued in a number of research and application strands, including Robotics, cyber-physical systems, and pervasive and ubiquitous computing. These systems must be able to modify their behaviour during execution to adapt to changing requirements and resource availability, computational constraints, component failure, network disruptions, and application-level circumstances.

Component-orientation is a highly popular, mainstream approach used to address these issues. Component frameworks operate by posing clear boundaries (in terms of provided and required service interfaces) between software 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 and dynamic binding between the components' interfaces. However, they fail to offer an adequate support for implementing adaptive systems, in terms of common adaptation models. Crucially, they also rely on pre-defined,

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

static components' and services' attributes, and offer only limited support for on-demand instantiation of components. All of these limitations make it 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.

The focus of this work is on the unification between agent, component and service concepts in a single methodology for the construction of autonomic software systems. On one hand, this work aims to produce autonomic systems by applying a well proven agent model to overcome the limitations of current component-based systems and thus create a set of re-usable and modular adaptation mechanisms. On the other hand, the same approach results in a modular and efficient agent platform grounded in mainstream component technology.

2 Related Work

Modern component frameworks provide service brokering mechanisms to enable run-time component composition. Most relevant to this work, OSGi [1] defines a lightweight component framework, which is used as a shared platform for network-provisioned components and services specified through Java interfaces.

The 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. The separation between services and their actual implementations is the key to enable system adaptation. With OSGi, developers can also associate lists of name/value attributes to each service, and use the LDAP filter syntax to search the services that match given search criteria. 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. The user may also assign a rank to each service, which can be used to describe its quality and importance, so that OSGI can automatically locate the highest-ranked implementation when queried about a particular service. A-OSGi [2] goes a step further by providing a number of mechanisms to create self-adaptive architectures, including plan monitoring, and context-sensitive execution of bundle adaptation steps.

This work shares many similarities with existing frameworks implementing autonomic control cycles, such as the Raimbow framework [3], and with workflow management systems, especially those designed for on-the-fly system composition, such as [4]. Compared to those initiatives, Self-OSGi injects agent-based mechanisms into the fabric of a typical component framework, de-facto transforming components into autonomous agents.

More recently, the Active Component [5] concept 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. However, such an initiative does not leverage mainstream component & service-based standards, such as OSGi. Noticeably, existing Java-based agent platforms, such as Jadex, 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 modularity enabled by the OSGi framework.

Compared to A-OSGi, Self-OSGi provides the ability to control the life cycle and measure the performance of single components rather than entire bundles, which enables the definition of fine-grained system's adaptation policies. In addition, Self-OSGi's design is shaped on the BDI agent model, discussed in the following section, thus leveraging well proven adaptation mechanisms to drive the dynamic instantiation and selection of components and services.

2.1 The BDI Agent Model

Kinny et al. [6], 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
- 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

Each plan 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-condition), and Ψ is the context for which the plan can be applied (the plan's precondition). 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 to AND/OR goal-plan execution trees.

An agent must rely on explicit representations of its own goals in order to keep track of goals achieved and yet to achieve, as done in modern agent toolkits, such as Jadex [7]. Goal events in BDI agents are usually posted by using special temporal operators like *achieve* (used to request the achievement of a new goal) and *maintain* (used to specify a homeostatic goal - one that must be re-achieved if it ever becomes unsatisfied).

The separation between goals and plans, and 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.

3 Component & Service-based Agents

Self-OSGi translates the BDI model outlined in the previous section into general component & service concepts, as illustrated in Fig. 1. In particular, the separation between the services' interface (called *service goals* in Self-OSGi) and the services' implementation (*component plans*) is the basis for implementing both the declarative and the procedural components of a BDI-like agent within a component & service framework such as OSGi.

The ***Self-OSGi Hook*** intercepts service requests made to the OSGi Service Registry. On its own, OSGi can only match the request's LDAP filter with pre-defined service attributes of already active components. However, the agent execution model requires the on-demand activation of component plans and their context-sensitive selection. For this reason, Self-OSGi obtains from the DS the XML description of all the available components. Within Self-OSGi, these descriptions may include pre-conditions expressed as LDAP filters over the attributes stored in special ***Belief Set*** components. Contrary to static service attributes, Belief Set attributes are updated, at run-time, by application components, to reflect some of their internal variables (e.g. agent's beliefs representing the perceived status of the environment).

Self-OSGi implements the BDI cycle by (i) finding all the component plans with satisfied pre-conditions, (ii) identifying the most suitable one by using user-provided, application-specific, ranking components. The latter have access both to the belief set and the statistics reporting components' performance during past activations. These are collected by interposing 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. Thanks to its

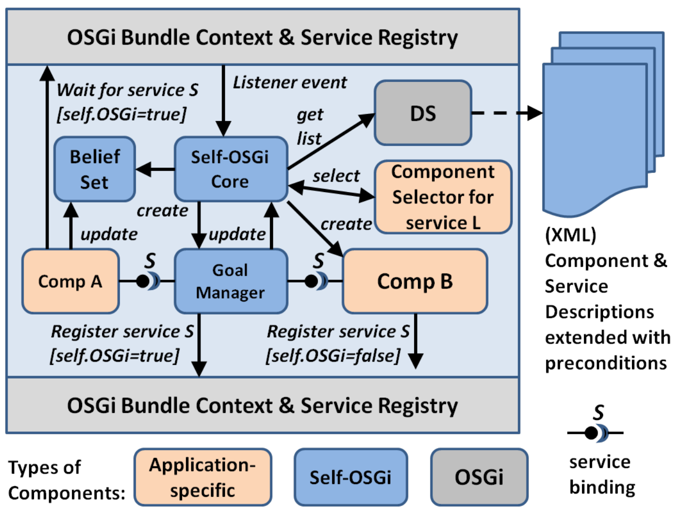

OSGi Bundle Context & Service Registry
Wait for service S [self.OSGi=true]
Listener event
get list
DS
Belief Set
Self-OSGi Core
select
Component Selector for service L
create
update
update
create
Comp A
S
Goal Manager
S
Comp B
Register service S [self.OSGi=true]
Register service S [self.OSGi=false]
OSGi Bundle Context & Service Registry
(XML) Component & Service Descriptions extended with preconditions
S
service binding
Types of Components:
Application-specific
Self-OSGi
OSGi

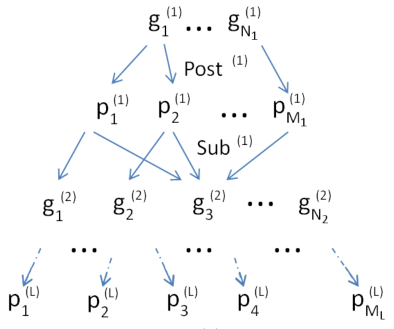
Fig. 1 Example of Self-OSGi system: Comp-A uses service *S* provided by Comp-B

mediation, a Goal Manager is also used to catch application exceptions and trigger the selection of an alternative component plan.

4 Evaluation

In order to assess Self-OSGi performance, a test domain, TD, is used to describe a typical Self-OSGi application where both the set of available component plans, P, and the set of service goals, G, are organized in L levels of abstraction (see Fig. 2.a): From main application goals to sensor and actuator component plans interfacing with the application and/or physical environment. TD also describes the structural coupling of the system components by defining the service goals they provide and require, respectively, indicated with *Post* and *Sub*, as well as any logical and/or temporal dependencies between them, as described by the set of equations (1).

$$\begin{aligned}
TD &= < L, G, P, Post, Sub > \\
G = \{G^k\}, &\text{ where } G^k = \{g_j{}^k\}, k = 1..L, j = 1..N_k \\
P = \{P^k\}, &\text{ where } P^k = \{p_j{}^k\}, k = 1..L, j = 1..M_k \\
Post = \{Post^k\}, &\text{ where } Post^k \in \{0,1\}^{N_k \times M_k}, k = 1..L \\
Sub = \{Sub^k\}, &\text{ where } Sub^k \in N^{N_k \times M_k}, k = 1..L \\
Post_{i,j}^k = 1 &\text{ iff } p_j{}^k \text{ achieves service goal } g_i{}^k \\
Sub_{i,j}^k \neq 0 &\text{ iff } p_j{}^k \text{ needs service sub-goall } g_i{}^k \\
(Sub_{i,j}^k < Sub_{h,j}^k) \rightarrow \ & g_i{}^k \text{ must be achieved before } g_h{}^k
\end{aligned} \tag{1}$$



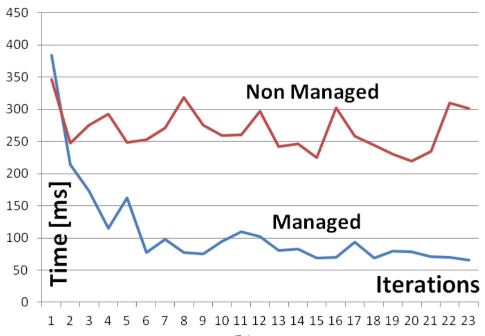


Fig. 2 (a): Structure of the layered test-bed, (b): results from a component-replacement test

The tests presented here were performed by setting the number of service goals in each layer, N_k,to 2^{k-1}, while the number of component plans was set to M_k=3N_k. The system was fully connected $(Sub_{i,j}^k = i, Sub_{i,j}^k = 1, \forall i, j)$. For instance, for $L = 5$, the system has 31 service goals and 93 component plans. With such a setup, the average overhead of 0.15ms imposed by the Self-OSGi service management (performance measurement, selection and proxy mechanisms) was measured by comparing the time needed to achieve the root service goal, measured for $L \in [2..5]$, with or without Self-OSGi.

Finally, Fig. 2.b shows the execution times obtained with $L = 2$ with component plans that require different (random) CPU times, and with a plan selection component assigning 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 plans at each iteration, ultimately converging on the best policy to achieve the root goal in the shortest time.

5 Conclusion

This paper has examined component, service and agent concepts, and has illustrated the design and the implementation of the Self-OSGi - ai framework for the construction of systems with Self-* properties built on OSGi Java technology. Future work will seek to exploit planning and learning techniques to tackle some of the main limitations of adaptive component & service frameworks, such as their lack of look-ahead capabilities and their reliance on hard-coded preconditions.

Acknowledgments: This work is partially supported by the EU FP7 RUBICON project (contract n. 269914) - www.fp7rubicon.eu.

References

1. OSGi: http://www.osgi.org/Main/HomePage Accessed 21 October 2011.
2. J. Ferreira, J. Leitao, and L. Rodrigues, A-OSGi: A framework to support the construction of autonomic osgi-based applications, Technical Report RT/33/2009, May 2009.
3. Shang-Wen Cheng, et. al, Evaluating the effectiveness of the Rainbow self-adaptive system, SEAMS, pp.132-141, 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2009
4. Louis Thomas, et. al, Achieving Coordination through Dynamic Construction of Open Workflows. Middleware 2009: 268-287
5. Lars Braubach, Alexander Pokahr Addressing Challenges of Distributed Systems using Active Components in: In Proceedings of 4th International Symposium on Intelligent Distributed Computing
6. Kinny, D., Georgeff, M., Rao, A. (1996) A methodology and modeling technique for systems of BDI agents. Proc. of MAAMAW96, LNAI1038, Springer-Verlag, p56-71, 1996.
7. Alexander Pokahr, Lars Braubach, Winfried Lamersdorf, A Goal Deliberation Strategy for BDI Agent Systems, MATES-2005, Springer-Verlag, Berlin Heidelberg New York, pp. 82-94.