Integrating IoT and IoS with a Component-Based approach

Grégory Nain, François Fouquet, Brice Morin, Olivier Barais, Jean-Marc Jézéquel

Integrating IoT and IoS with a Component-Based approach*

Grégory Nain[1], Francois Fouquet[2], Brice Morin[1], Olivier Barais[2], Jean-Marc Jézéquel[1,2]
[1] INRIA, Centre Rennes - Bretagne Atlantique, [2]IRISA, [2]Université de Rennes 1
Campus de Beaulieu, 35042 Rennes, France
Email: {Prenom.Nom}@irisa.fr

Abstract

There is a growing interest in leveraging Service Oriented Architectures (SOA) in domains such as home automation, automotive, mobile phones or e-Health. With the basic idea (supported in e.g. OSGi) that components provide services, it makes it possible to smoothly integrate the Internet of Things (IoT) with the Internet of Services (IoS). The paradigm of the IoS indeed offers interesting capabilities in terms of dynamicity and interoperability. However in domains that involve "things" (e.g. appliances), there is still a strong need for loose coupling and a proper separation between types and instances that are well-known in Component-Based approaches but that typical SOA fail to provide. This paper presents how we can still get the best of both worlds by augmenting SOA with a Component-Based approach. We illustrate our approach with a case study from the domain of home automation.

Acknowledgment

*The research leading to these results has received funding from the European Community's Seventh Framework Program FP7 under grant agreements 215483 (S-Cube, http://www.s-cube-network.eu/) and 215412 (DiVA, http://www.ict-diva.eu/).

1 Introduction

Building easily configurable applications for house automation, e.g. in the context of Ambient Assisted Living (AAL), is a complex undertaking because it has to marry the ever evolving needs of the customers with the diversity of devices, appliances, communication links, communication protocols available in this domain.

There is thus a growing interest in leveraging Service Oriented Architectures (SOA), with execution platforms such as OSGi that make it possible to smootly integrate the Internet of Things (IoT) with the Internet of Services (IoS). The paradigm of the IoS indeed offers interesting capabilities in terms of dynamicity and interoperability. For instance, we developed EnTiMid [15] as a service oriented framework to cope with compatibility problems in the domain of house automation, as well as for managing systems (devices, versions, communication links) and their configuration in an easy and unified way.

However typical SOAs fail to provide the loose coupling and proper separation between types and instances that are needed in domains that involve "things" (e.g. home automation). For instance two light appliances may offer the same *type* of service (turning light on and off) but different *actual* services, if only because they are located in different rooms. These loose coupling and proper separation between types and instances are however well known in Component Based approaches.

Besides, IoT-based applications are also characterized by the fact that things and their associated software are introduced or removed from an execution environment at runtime. Supporting this degree of dynamism is usually done programmatically, and our experience intends to simplify this task and to provide developers with an explicit view of the system architecture, while supporting its dynamic evolution. This paper presents how we can get the best of both worlds by augmenting an OSGi-based SOA with a Component Based approach.

The rest of this paper is organized as follows. An investigation on existing tools and approaches in component and services domain is conducted in section 2, and leads to a list of requirements for integrating service based approach and component based approach for IoT system. Then our contribution using component-based approach on a service-based execution environment is detailed, illustrated and evaluated in section 3. Section 4 places our contribution with relation to other approaches close to ours. Section 5 concludes and highlights future work.

2 Existing tools and approaches

As a preliminary work, this section investigates on tools and approaches already developed in both components and service worlds. Using an IoT / IoS integration vision, pros and cons of each platforms are listed to better understand and specify the needs for a new platform getting the best of two paradigms for IoT/IoS system.

2.1 In the services domain

Java Business Integration(JBI) or JSR 208 is an industrial Java standard that eases the software integration over a Service-Oriented Architecture. Its goals are to avoid specific developments and to allow a reuse of Java technologies such as WebServices, BPEL, JMS. OpenESB by SunMicrosystem, ServiceMix by Apache Foundation or PeTaLs by OW2 are mature projects that comply with JBI specifications. This standard defines a component model on top of an Enterprise Service Bus (ESB).

Enterprise Service Bus refers to a business middleware family built around the SOA paradigm. These middlewares provide a runtime environment for deploying business ser-

vices. They offer a mean to integrate legacy software as services into the business service orchestration. Every service is declared within the scope of the ESB runtime, which acts as the only mediator of services in the enterprise.

The JBI component model defines components with independent life-cycle, that communicate through their services over a normalized message middleware. This middleware acts as an abstraction layer for communications, and facilitates the integration of legacy software. According to their function, JBI components are split into two categories:

- *Service Engine Components* are directly hosted by the JBI runtime environment and cannot communicate outside of this scope. They are in charge of message processing, routing or orchestrating services.

- *Binding Components* expose or consume standard JBI services and perform the bindings with external non-standard software.

The component framework also describes a packaging for components. Service descriptions are encapsulated into Service Units, which are then encapsulated into deployable business component called Service Assemblies.

Besides the good properties in terms of interoperability offered by the message middleware and in terms of openness with the *Binding Components*, there is no clear separation between types and instances of services/components. Moreover, no introspection of services is offered and the interconnections between components, are not explicitly expressed, and sometimes, even hard-coded inside the components.

The OSGi Service Platform provides functionality to Java that makes Java the first environment for software integration and thus for development. The OSGi technology provides standardized primitives to construct applications from small, reusable and collaborative components. These components (also called Bundles) can be composed into an application and deployed.

The OSGi Service Platform makes it possible to dynamically change the composition of bundles with no need to restart the application. The OSGi technology provides a service-oriented architecture to minimize and to manage the coupling between bundles, enabling the components to dynamically discover each other for collaboration.

A service is specified by a Java interface. Bundles can implement this interface and register the service with the Service Registry. Clients of the service can find it in the registry, or react to it when it appears or disappears. This is similar to the service-oriented architecture made popular with web services. The key difference between web services and OSGi services is that web services always require some transport layers, which make them much slower than OSGi services, which may use direct method invocations. Also, OSGi components can directly react on the appearance and disappearance of services.

The mechanisms provided by OSGi to manage the coupling are not sufficient in the context of IoT. Since the coupling is explicitly hard-coded inside the bundles, any change to the service discovery and the binding policies implies replacing the bundle. Indeed, the application architecture is never made explicit and the OSGi platform only offers very limited introspection primitives.

2.2 In Components domain

Fractal [2] is a modular and extensible component model to design, implement, deploy and reconfigure various systems and applications. Famous implementations of Fractal are Julia and AOKell (Java), Cecilia (C), FractNet (.NET) and FracTalk (SmallTalk).

The Fractal component model supports the definition of primitive and composite components. Each Fractal component consists of two parts: a controller which exposes the component's interfaces, and a content which can be either a user class or other components in composite components. The model makes explicit the bindings between the interfaces provided or required by these components, and hierarchic composition (including sharing).

Primitive components contain the actual code, and composite components are only used as a mechanism to deal with a group of components as a whole, while potentially hiding some of the features of the subcomponents. Primitives are actually simple, standard Java classes (in the Java distributions of Fractal) conforming to some coding conventions. Fractal does not impose any limit on the levels of composition, hence its name.

All interactions between components pass through their controller. The model thus provides two mechanisms to define the architecture of an application: bindings between interfaces of components, and encapsulation of a group of components into a composite. By default, Fractal proposes 6 controllers that may be present in components: Attribute, Name, Binding, Content, Lifecycle and Super controller.

Reflective execution platforms like Fractal or OpenCOM [1], do not provide a clear distinction between the reflection model and the reality. Modifying the reflection model implies modifying the reality: there is no mean to preview the effect of a reconfiguration before actually executing it, or to execute what-if scenarios to evaluate different possible configurations, etc. This lack of an explicit and independent reflection model require to perform most of the verifications (*e.g.* pre-condition on reconfiguration actions, as proposed by Léger [12]) during the reconfiguration process itself, and roll-back if it encounters a problem.

In addition, component models as Fractal are a bit opaque with respect to the outside world, making the opening and

reuse by third party applications complicated if not foreseen in advance. At last, the dynamicity of an application running over Fractal is compromised because the deployment of new components can not be done at runtime without restart.

2.3 Requirements for component for the Internet of Servcies

As a conclusion, this sub-section summarizes the benefits of both worlds (SOA and CBSE) and outlines the requirements an execution environment fully adapted for IoT and IoS integration must comply to.

Rq1: *An explicit and independent reflexive model of the architecture living at runtime*. Reflecting the actual application, the model makes it possible to reason about the application state. Then an adaptation engine is able to select, test and validate an adaptation scenario on the model, before actually performing the adaptation on the running system [12].

Component-based execution systems often offer introspection capabilities making it possible to build a model view of the running system. SOA execution platforms do not have such an ability.

Rq2: *Components coupling managed from outside* For the components to be highly independent, they must not embed any dependency resolution mechanism. Moreover, this extraction would make it possible to modify the resolution policies, or change the connections to adapt the system with no need to deal with business components. Having a clear and explicit description of the relations between components gives a better understanding and makes the analyze of the system much more accurate and so, leads to better adaptation decisions.

The component connections are often explicit in Component-Based systems, but are never in SOA, and dependencies resolutions are even hard coded in services.

Rq3: *Interoperability and opening to the outside world* is an essential principle in IoS. The goal is to offer a service in a standardized way to any other system that would like to use it. Even if the system is managed as a component based application, any third party application must be able to use the services offered by the managed devices (IoT speaking) and more generally components. Services must thus be exposed as classical services, while their 'component like' management should remain hidden.

This is natively offered by SOA using interfaces and registries to exposes services to the world. Component-Based applications are in the other hand, living in their close world.

Rq4: *Hot deployment* ability is absolutely necessary to ensure future evolutions and adaptations to the protocols and devices. The execution platform must support dynamic deployments and adaptations of the application during runtime with no restart.

SOA consider that services appear and disappear at any time. The hot deployment is thus essential and natively taken into account. Component-based applications do not address this concern.

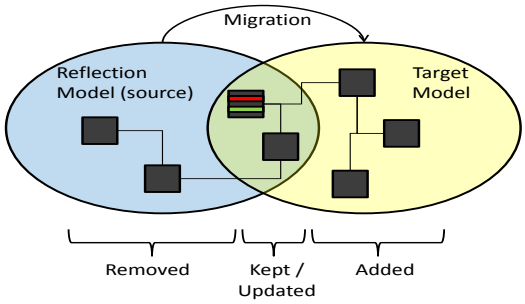

Figure 1. Identifying Differences between the source and the target configurations.

Rq5: *Minimize the adaptation time* Another strong constraint working with 'things' is that the entire time of realization of an action starting from the moment a person acts on a sensor, to the moment something happens, must be less than 250 milliseconds for it to be considered as immediate by a human person. The reconfigurations of the system must fit within this constraint, or more specifically, the transition time from a stable configuration to another one should not exceed this limit.

3 Description of the solution

Models At Runtime technics are used to address the first requirement. An engine capable of reasoning on the runtime model manages the connections and deployments of components. The wrapping of components into bundles makes it possible to register services from a component and open it to the world as required in *Rq3*. Then the underlying OSGi platform natively supports the *Rq4*.

The next paragraphs describes our contribution in more details through different point of view highlighting our answers to the requirements. A little example and an evaluation of the tool are presented on follow.

3.1 Solution in details

Models at Runtime

In order to ensure reflexivity(*Rq1*), the system keeps a model of the actual running objects/components and their connections (bindings) at any time. Configurations, reconfigurations and adaptations of the system are handled as follow.

Model creation In the cases of a first configuration or a reconfiguration of the system, the target model is given to the system by a human person. For an adaptation, the target model is automatically derived from a Dynamic Software Product Line model [4, 13, 10, 8]. This derivation is driven by a reasoner component which selects the features most adapted for the current context. This reasoner then derives the corresponding architectural model using model composition techniques. This model (if valid) is stored in a cache, which is managed according to a standard caching algorithm.

Online Validation This step relies on invariant checking: for all the produced configurations, we check that all the

invariants are satisfied. The open-source Kermeta metamodelling language [14] is used to manage different checking strategies and check all the invariants (expressed using an OCL-like syntax). We distinguish between two types of invariants: generic and application-specific. Generic invariants can for example check that all the mandatory client ports are bound to compatible server ports. Application-specific invariants can for example check that the EnTiMid application always has a communication component.

Component coupling managed from outside

Identify and validate the changes (Fig. 2, Step A) After validation, the first task is to identify the differences between the model representing the running system (source model) and the target model the system must switch to, as illustrated in Figure 1. During the comparison, the next 7 types of primitive commands can be found. *1.* **start** and **stop** components. *2.* **add** and **remove** components. *3.* **add** and **remove** bindings. *4.* **update** components.

These primitive commands represent atomic differences between the two configuration models. To allow the change of the component management policy, the comparison system only deals with abstract commands. This advocates for *Rq2* by decreasing coupling. The real commands are instantiated (not yet executed) according to the actual policy, during the model comparison.

These commands are stored in a collection and ordered according to an heuristic [11, 16] that ensures a safe migration from the current to the target configuration. Before actually executing the commands, the list is parsed to verify that all the commands can be executed. For example, for all *AddComponent* commands, the presence of the specific component factory is checked to ensure all components can actually be added without problem. Doing this kind of verification for all commands ensures that the command execution will properly execute. If a command is detected as non executable, a report clearly describes the problem, and no command at all is executed. This way, the system is always kept consistent.

Because of an adaptation, some links (bindings) between components may appear or disappear, for the system to act differently. In case of classic components, adding or removing bindings is as simple as setting or unsetting a variable. Generally, a component missing one mandatory binding is stopped because it cannot run any longer. However, in the case of service based systems such as EnTiMid, the component may still offer its services to third-party applications and thus should not always be stopped. In other words, a "light component", virtual representation of a real light, may not be bound to any other component, but might still serve another application to control this light.

Other behavioral constraints can require more complex actions that just a set or an unset. For instance, if an alarm has been triggered, and if the user does not process this alarm, the system must be able to propagate the information anywhere else for the alarm to be treated. If the communication link is asked to be removed, this is structurally correct, but it may be part of an operation being treated, and so it does not have to be removed immediately.

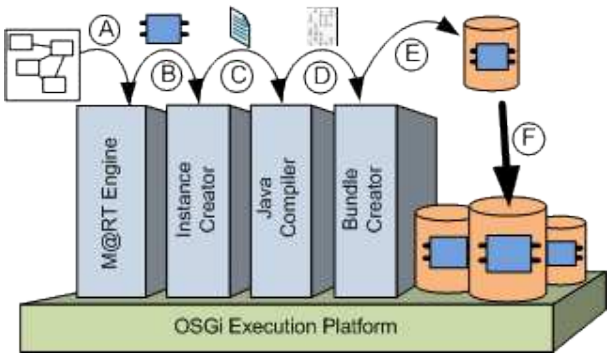

Figure 2. Overview of the toolchain

Two component families, for transparency and opening

In classical component architecture, the component assembly is made by an entity, and the components are not available from external applications. Working on a service-based platform implies that the system takes into account the services registrations and unregistrations. EnTiMid has been developed to allow third-party applications to access devices in a unified way, whatever the vendor, through different protocols. Managing all the devices as pure components hinders the interoperability and access with/from third party applications. To tackle this issue, we divide components into two families: *functional components* and *device components*.

Functional components are designed to be as light as possible to reduce the transition time *Rq5*. They only exist in memory and cannot be accessed from the outside. They cannot publish services for third-party application to use them. They are abstract components (such as timers, event publishers, parallelizers, sequencers, etc.) used to obtain a specific behavior, or to connect components in different ways than a simple binding.

Device components, on the other hand, support the requirement *Rq3*. They wrap components standing for real-life physical devices (lights, switches, alarm sensors, weather sensors, etc.) into on-demand generated OSGi bundles. The start of a device component is made into two steps. Since the component is contained in a bundle, the bundle must be started first (and stopped last). When starting, the bundle creates the component instance, sets global properties and publishes the services needed for the component to be manageable by the M@RT Engine. In a second step the actual component starts. After all the needed variables have been set for the component to run properly, it publishes its services on the OSGi context for them to be used by other applications.

The wrapping of device components and their hot deploy-

ment (*Rq4*) is made using a chain of actions. This chain is presented in Fig. 2 and explained in the following.
Step A shows the model of the new configuration has to reach. This step starts the chain.
Step B The Model@Run-Time Engine asks the *Instance Creator* to embed a new component instance into a new bundle. To do so, the *Instance Creator* generates an activator and a manifest, according to the instance informations given by the M@Rt engine. The Manifest, like any classic OSGi bundle manifest, gives information about the packages needed for the bundle to run, and which are the new provided packages. The role of the activator is to ask a factory for a new instance of the device at bundle start. Then, it registers the component as a service implementing the component type. The properties of the service registration gives the instance name for the system to be able to find the components. The activator is a Java class that needs to be compiled before being included in the bundle.
Step C The generated Java file is then given to an Eclipse Java Compiler (ECJ) embedded on the platform. After linking all the libraries, the compiler produces Java compiled files.
Step D consists in creating the actual bundle to be deployed on the platform. All the files and necessary resources are packaged and returned.
Step E On return, the bundle is first saved in a local repository for it to be available in case of a future reconfiguration, avoiding a new generation step.
Step F The bundle is finally installed on the running OSGi platform.

3.2 Application in the context of Ambient Assisted Living

The simple example presented here has been extracted from a study conducted in collaboration with elderly people in the context of Ambient Assisted Living. The EnTiMid application is running on a MSI Wind[1] equipped with a touchscreen, adapted to elderly people.

p1:Parallelizer is a functional component, which is only living in memory and is not embedded into a bundle. The parallelizer *p1* is not available from the outside (from third party applications). Its role here is to take part in the application logic by simultaneously launching the execution of the connected components. *bedLight:TYXIA_510* is a device component, which must be available for third party applications. It controls the light placed at the head of the bed. This component is embedded into a bundle to be able to expose a service to the world, via the OSGi registry. Any other application running on the OSGi execution platform can thus access and use this bundle to control the light. A service query on this type of component results in a real action on the actual device in the house (switch a on the light for example).

[1]CPU: Intel Atom 230@1.6GHz, RAM: 1Gb and OS: Ubuntu 9.10 (Linux kernel: 2.6.31-17-generic)

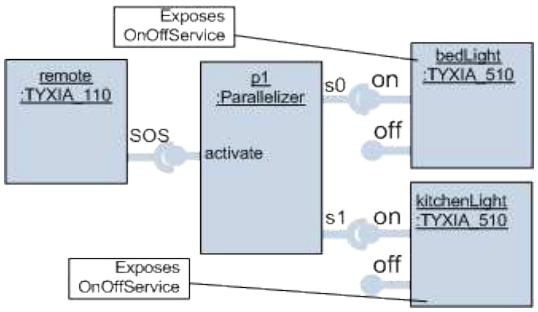

Figure 3. Simple example of architecture

At this point, it is interesting to note the clear separation of component type/implementation/instance. There are only two component types *Functional components* and *Device components*. These components can have multiple implementations: Parallelizer, Sequencer, EventPublisher, for functional components, and TYXIA_510, RMG4S, Nabaztag for device components. At last, multiple instances of each component can exist at runtime, because each component is set to be connected to a specific device or to have a specific behavior. *bedLight:TYXIA_510* and *kitchenLight:TYXIA_510* are clearly illustrating that.
The availability of services on the OSGi platform opens new possibilities. Anybody familiar with OSGi developments is then able to create his own plugins to control the devices of his house. Let's illustrate this with an example exposing devices through an interactive GoogleTalk instant messaging conversation. XMPP is an instant messaging protocol. This protocol is used by various instant messaging providers (e.g. Jabber, GTalk). In this context, the bundle exposing devices on GTalk is considered as a third-party application, running on the OSGi platform. Using an interactive question/answer robot, one can manage his home (through EnTiMid) from his computer. This way, it is possible to switch on/off the TV, the lights, the heaters, etc from your office or from your smart-phone, exactly as if you were discussing with your best friend on GTalk. The XMPP bundle simply lists (on demand) all the EnTiMid services published on the platform, and interactively describes the available things you can do. It is interesting to note that the XMPP developer does not need to know how the components are managed to be able to create an EnTiMid third-party application. The only thing to know is how OSGi works, and that EnTiMid services are published as org.entimid.services.* on the OSGi service registry. Everyone can then develop his own plugins and act on the house by using EnTiMid provided services.

3.3 Evaluation of the solution

The evaluation consists in executing a small but realistic scenario, and recording the results. Among the various services offered by EnTiMid, we concentrate on the alert system. This system offers an elderly person a mean to throw help requests using a very simple device. The system considered can be in either of two states.

During the day, the system is composed (amongst other components) of a simple remote control (*TYXIA_110*) and a parallelizer, connected as shown in Fig. 3. The parallelizer on its side, is connected to two components not presented in the figure. The first is an *SMS Sender* component which is in charge of sending a pre-established text message to an emergency service. In parallel, the second component *Nabaztag* (text-to-speech) informs the person that the request for assistance has been considered.
In summary, the alert system is composed of four components during the day.
At night, in addition to the four components of the day, two more components (*bedLight* and *kitchenLight*) are present on the system. This way, the house is lighted in case of emergency. The components are connected to the parallelizer as presented on Fig. 3, at the same level as the *SMS* and *Nabaztag* components.
Let describe the execution scenario:
Initial Deployment: the system is deployed during the day, and configured to meet the requirement of the elderly person living in the house. In this configuration, the elderly person can send emergency messages via SMS to a control center by using a simple device with one button (Tyxia 110). While the SMS is sent, the person is notified via a device equipped with text-to-speech capabilities (Nabaztag). In this configuration, the complete EnTiMid system is composed of 18 components (bundles or simple components in memory) and 9 bindings among these components.
Night: when the night falls, or more precisely when the light detector sends values lower than a given threshold, the application is reconfigured. The emergency feature is still active and switches to the *night configuration*. Then, the system is composed of 20 components and 18 bindings among these components.
Day: when the day rises, the system is reconfigured into the *day configuration*.
The Fig. 4 presents a sequence of reconfigurations of the system. After the initial deployment (*State 1*), we iterate night states (*State 2* and *State 4*) and day states (*State 3* and *State 5*) during the next two days.
During the initial configuration, all the components needs to be deployed. In particular, all the components that need to be wrapped into OSGi bundles have to be compiled and packaged at runtime. This explains the rather long reconfiguration time of step 1: 2.5 seconds.
During the first reconfiguration (day → night, step 2) all the already deployed components are reused. In addition, the *bedLight* and *kitchenLight* have to be compiled, packaged, properly deployed and connected to other components. This is realized in less than 400 ms.
The next 3 reconfiguration (night → day → night) are much faster. Step 3 simply consists in unbinding and removing the 'light' component. Step 4 is similar to step 2. However, we maintain a cache of pre-generated components (jar files). The 'light' component is thus directly reused, with no need to compile and package it. Step 5 is similar to step 4. The actual reconfiguration time of these steps is less than 100 ms.

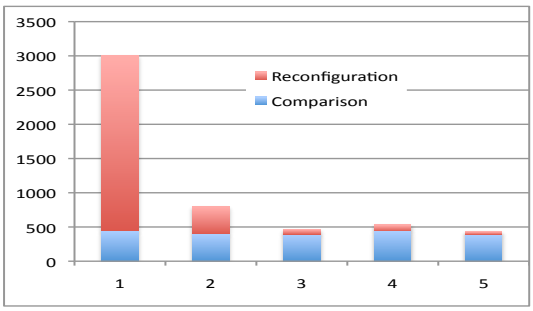

Figure 4. Time (in ms) spent in Configuration Comparison and Actual Reconfiguration

For each reconfiguration, we first perform a model comparison. This model comparison takes an almost constant time of 400 ms to compare models with about 20 components. This comparison step is executed before the actual reconfiguration to plan the reconfiguration (*i.e.* instantiate the reconfiguration commands). It thus delays the actual reconfiguration of the system but does not impact the (un)availability of some components during dynamic reconfiguration. This way, the *Rq5* about minimizing the reconfiguration time for it to be unperceived by the user is fulfilled.

3.4 Discussion

Our Model-Driven dynamic adaptation process is obviously less efficient (*w.r.t.* to the time needed to actually reconfigure an application) than hard-coded reconfiguration scripts executed on platforms like Fractal, for several reasons:
A global model comparison is systematically performed between the source and the target configuration to determine all the actions needed to reconfigure the system.
The wrapping of some components (embedded into OSGi bundles) is time consuming, because of the compilation, packaging and deployment.
However, these two drawbacks have significant advantages, and can easily be minimized:
The model comparison discharges developers from *writing low-level and error-prone reconfiguration scripts*. The system automatically computes a safe reconfiguration script that takes care of the life-cycle of components. It is important to note that this model comparison does not make an overhead on the actual dynamic reconfiguration time, it simply delays the reconfiguration. In other words, it does not impact the availability of services. Moreover, this model comparison could be performed by a third-party system (more powerful than a MSI Wind), which would return a list of reconfiguration actions. Indeed, each configuration is serialized in about 30 Kb (only a few Kb if zipped) so that it can be quickly transmitted on a network.
The OSGi components cache significantly improves the performances of the reconfiguration process seen in our exper-

iment. This cache of OSGi components can easily be initialized by a component that periodically iterates on a set of configurations (*e.g.* every 5 or 10 seconds during ldle time), and generates the necessary bundles to reconfigures the application with each configuration. This way, it is possible to initialize the cache a few minutes after the installation of EnTiMid. Moreover, the compiling and packaging of a component that would not be present in cache could also be externalized to a third party system, with more computation capacities, that would simply return the jar file of the required component.

4 Related works

Besides the tools listed in section 2, the approaches described in the following clearly appears as alternative solutions. Indeed, this section aims at positioning our proposition with relation to these.

BluePrint. OSGi describes a dynamic component framework. This framework provides low-level mechanisms for implementing modular and dynamic applications. OSGi is already part of many applications, mature implementation are Felix by the Apache foundation or Equinox by the Eclipse Foundation. Blueprint container was added in the OSGi 4.2 specification to define a dynamic dependency injection mechanism on top of OSGi components. It is highly inspired by Spring Dynamic Module or Gravity [3].

This framework allows managing the dynamic nature of OSGi applications through a declarative XML-based service mechanism. Instead of directly using the OSGi packaging, BluePrint uses POJO objects, whose services and dependencies are declared in an XML file mainly inspired by Spring DM [2] and iPojo [6]. Based on this description, the container instantiates new components at runtime and properly binds them together by injection. Inversion of Control [7] imposes a clear separation between components. BluePrint makes it possible to separately declare hosted and required services and their implementations, thus reducing the complexity of the code. BluePrint Container monitors the OSGi platform and reacts to event like Bundle discovery or Bundle shutdown. With this introspection the container can dynamically declares new bundle and resolve dependencies with an equivalent service when a component is shut down.

BluePrint and our solution are very close in many points. Both of them can realize adaptations at runtime to keep the system in conformance with given constraints. In our solution, the constraints are solved at a business level, by working on high-level models of the running system. BluePrint simply solves technical needs in term of mandatory services. When a service fails, BluePrint replaces it by another one, with no overview on the changes this action can imply. This may lead to an unstable application, because the new deployed service may require other new services to run, and conflicts can appear. Working at a higher level of abstraction can help finding these unwanted conflicts and avoid them by selecting the most appropriate available service.

The resolution mechanism of BluePrint, its dynamicity offered by the event listener mechanism, and the use of XML declarative files instead of compiled code, can really be helpful in our case, but a development effort is needed for BluePrint to be able to load and work with *completely specified models*. In the future, we will probably try to integrate it, and measure how efficient the BluePrint resolution is, compared to our implementation.

The main benefit of our proposition is the fact that the component model is still available at runtime, consequently we obtain a reflexive component runtime for OSGi. This idea of models@runtime can ease the implementation of runtime model checking or the implementation of the adaptation layer.

SCA. SCA is a standard specification that provides a component model for Services Based Applications. It has several advantages: First it decouples the application business logic from the details of its invoked service calls. It supports a multitude of programming language for the component and services implementation. The ability to seamlessly work with various communications mode (asynchronous, MOM, RPC). It provides several types of binding to easily interact with legacy components or services providing access by technologies such as Web Services, EJB, JMS, JCA, RMI, RPC, CORBA and others. The value proposition of SCA is to separate the implementation of services and the wiring logic (Assembly model) of a service based application. The overhead of business logic programmer concerns regarding platforms, infrastructure, plumbing, policies and protocols are removed. Indeed, assembly model offers a mean to provide quality of service features for security or transactions. This specification is implemented in Frascati by OW2 or in Tuscany by the Apache Foundation.

SCA takes benefits from research experience and industrial experiences in building a reflexive component model. As a component model, we could reuse this architecture description language. For the integration of IoT and IoS, it misses the component implantation life-cycle management. Consequently, in the SCA specification, a component implementation is not a runtime artefact that can be easily deploy, undeploy, migrate etc. On that issue, OSGi specification provides the concept of bundle and a management layer for bundle which is highly valuable for IoT systems. Software to manage plug'n play device should be hot-deployed and removed when a component appear or disappear. Our proposition reuse this concept of bundle for managing component implementation and component instances. Consequently, component implementation and component instances can be managed uniformly. As a result, our work can be seen as an experience of taking the best of SCA concepts (a reflexive component model for SOA) and OSGi concepts (a Dynamic Module System for Java).

[2] http://www.springsource.org/osgi

Other approaches. Providing an ADL on top of OSGi is an issue that have been addressed by several works [6]. For example, Cervantes et al presents in [9] an approach to describe dynamic sensor-based applications using a declarative language called WADL. As IoT system dynamic sensor-based applications are characterized by the fact that measurement producers (sensors) and consumers are introduced or removed from an execution environment at runtime. Supporting this degree of dynamism is usually done programmatically, and the WADL intends to simplify this task and to provide developers with an explicit view of the system architecture, while supporting its dynamic evolution. They also show in [5] how a scripting language can be used to reconfigure the running system. WADL addresses mainly the producer/consumer interaction type between components. Indeed, WADL does not managed Component Assembly and Component Instances (wireapps) uniformly.

5 Conclusions and perspectives

Building easily configurable applications for house automation, e.g. in the context of Ambient Assisted Living (AAL), is a complex undertaking because it has to marry the ever evolving needs of the customers with the diversity of devices, appliances, communication links, communication protocols available in this domain. The paradigm of the IoS indeed offers interesting capabilities in terms of dynamicity and interoperability, but fails to provide the loose coupling, a proper separation between types and instances, and mechanisms to deploy and manage services, that are needed in domains that involve things.

After a study of different tools existing in service-based and component-based domains, we identified a list of requirements for an execution environment to be fully adapted to the integration of Internet of Things and Internet of Services.

To meet the identified requirement, a new tool having an explicit and independent reflexion model of the architecture living at runtime has been created. In this model, "things" and "services" are managed the same way to enforce the interoperability and the opening to the outside world of IoT applications. Moreover, components and the application behavior are managed from outside of the components offering the system a great flexibility. Then components can be deployed or removed with no restart of the system making any change or adaptation transparent for users.

Home automation and more generally the domain of human-machine interactions, considers the time for an action to be perceived as immediately realized must be less than 250 milliseconds. Our experiments show that the system we create is not far from this limit and may require more investigation on alternative solutions to gain on time.

References

[1] G. Blair, G. Coulson, J. Ueyama, K. Lee, and A. Joolia. Opencom v2: A component model for building systems software. In *IASTED Software Engineering and Applications*, USA, 2004.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The FRACTAL Component Model and its Support in Java. *Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.

[3] H. Cervantes and R. S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *ICSE'04: 26th International Conference on Software Engineering*, pages 614–623, Washington, DC, USA, 2004. IEEE Computer Society.

[4] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. *Computer*, 42:37–43, 2009.

[5] D. Donsez, K. Gama, and W. Rudametkin. Developing adaptable components using dynamic languages. In *35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEEA) 2009, August 27-29th, 2009*, 2009.

[6] C. Escoffier, R. S. Hall, and P. Lalanda. ipojo: an extensible service-oriented component framework. *Services Computing, IEEE International Conference on*, 0:474–481, 2007.

[7] M. Fowler. Inversion of control containers and the dependency injection pattern, January 2004.

[8] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *IEEE Computer*, 41(4), April 2008.

[9] L. T. Humberto Cervantes, Didier Donsez. An architecture description language for dynamic sensor-based applications. In *Proc. of 5th IEEE Consumer Communications and Networking Conference (CCNC) 2008, Las Vegas, Janvier 2008.*, 2008.

[10] P. Istoan, G. Nain, G. Perrouin, and J.-M. Jézéquel. Dynamic software product lines for service-based systems. In *9th IEEE International Conference on Computer and Information Technology*, Xiamen, CHINA, octobre 2009.

[11] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11):1293–1306, Nov 1990.

[12] M. Léger, T. Ledoux, and T. Coupaye. Reliable dynamic reconfigurations in the fractal component model. In *ARM '07: Proc of the 6th international workshop on Adaptive and reflective middleware*, pages 1–6, Newport Beach, CA, 2007. ACM.

[13] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.

[14] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Jamaica, octobre 2005. Springer.

[15] G. Nain, E. Daubert, O. Barais, and J.-M. Jézéquel. Using mde to build a schizofrenic middleware for home/building automation. In S. B. . Heidelberg, editor, *Towards a Service-Based Internet*, volume 5377/2008 of *Lecture Notes in Computer Science*, pages 49–61, Madrid, Spain, décembre 2008. IRISA/INRIA/University of Rennes1.

[16] J. Zhang, B. H. C. Cheng, Z. Yang, and P. K. McKinley. Enabling safe dynamic component-based software adaptation. In R. de Lemos, C. Gacek, and A. B. Romanovsky, editors, *WADS*, volume 3549 of *Lecture Notes in Computer Science*, pages 194–211. Springer, 2004.