Managing Multiple Applications in a Service Platform

Jacky Estublier, German Vega

Managing Mutiple Applications in a Service Platform

Jacky Estublier, German Vega
Grenoble University. LIG.
F-38041 Grenoble, France
{Jacky.Estublier, German.Vega}@imag.fr

***Abstract*—Information hiding and hierarchical decomposition are the corner stone of Software Engineering best practices. These principles have been applied in methods, architectures, programming languages, and run-time platforms. It is therefore a big surprise to notice that the recent dynamic service platforms, like OSGi, do not make use of these principles. In OSGi, all services are visible; a client asking for an interface will be wired to any service, randomly selected and implementing that interface,which makes almost impossible protection and encapsulation. Nevertheless, OSGi is very successful for its almost unique capability to support dynamicity; and because the current practice is to run a single application per platform. Unfortunately, the future of gateways, like OSGi, is to manage the "discovery", access and control of resources (logical as well as physical (sensors, devices)) shared by many applications. In the near future, OSGi will have to scale from a light weight mono-application gateway to a full-fledged dynamic platform. We have developed a layer on top of OSGi called APlication Abstract Machine (Apam) which provides OSGi dynamic capabilities, but also introduces a composite concept allowing multiple applications to cover the range isolation/collaboration from "black-box" (information hiding and hierarchical decomposition) to "scrambled eggs" as in service platforms, and through a variety of grey and white boxes with variable degrees of collaboration, sharing and control.**
The paper presents the state of practice, the challenges future dynamic platforms have to address, and how the Apam platform provides a solution to these issues. An assessment of the first Apam experimentations concludes the paper.

***Keywords*—service; service platform; service platform; dynamic application; encapsulation; sharing; protection; composite**

I. INTRODUCTION

If there is something is to retain from the last 40 years of software engineering it should be the principles of information hiding and hierarchical decomposition. Their systematic application avoided the disaster, announced since the 70's, of being unable to manage large programs (i.e. larger than a few thousands lines of code!).

Information hiding stipulates that a piece of code (that we will call a component) must publish (make visible) only the information strictly needed to use it (that we will call an interface). All the remaining: internal variables, code, methods and so on must be hidden from the component's users. It allows the component to evolve (improvements, bug fixes, additional features if published in other interface(s)) without impact on its users, as long as the interface is unchanged.

Hierarchical decomposition stipulates that a component can be made of other components; these inner components being hidden from external users using the encapsulation principle. Therefore a composite component cannot be discriminated from an atomic one. This principle allows scalability; at a given level, the apparent complexity of the system is only relative to the number and complexity of the components "visible" at that level.

These principles are pervasive and can be found, in different flavors, in methods, architectures, programming languages, and run-time platforms. After all, this is no surprise since these principles are universal and are adopted in all fields.

It is therefore a big surprise to notice that the recent dynamic service platforms, like OSGi [1], do not make use of these principles. It is true that a service applies the information hiding principle: a service has an interface and hides its content, but there is no hierarchical decomposition and no composite service.

Indeed, service composition is primarily achieved by orchestration [2]. An orchestration aggregates the functionality of several services and is itself published as a service, so both principles seem to be satisfied. Unfortunately, an orchestration does not hide (encapsulate) the services used; it creates a new higher level service, but it does not reduce complexity since it adds one service in the system, and removes none.

In traditional hierarchical decomposition, at each level of abstraction the number of visible components is relatively constant so that the perceived complexity at each level of abstraction is kept relatively constant. In contrast, with current service composition approaches when new higher level services are created, the number of registered services increases; there is not a new level of abstraction and the perceived complexity keeps increasing.

Similarly, in hierarchical decomposition the composite object provides a context and scope for resolving the required dependencies of its constituents. In contrast, in service oriented platforms, like OSGi, when a client, at runtime, asks the service registry for an interface "A", it will be bound to any randomly selected published service providing "A" and satisfying the request, without considering the context of the client, and irrespective of who published "A", and for which purpose. If no service "A" exists, the platform simply returns "null" to the client.

This brutal, simplistic and non-deterministic behavior is (apparently) in strong opposition with the experience and the best practices accumulated during the past decades.

Despite this apparently unacceptable behavior, service platforms are very successful. This is because, in practice, a platform supports a single application made of components of similar granularity or that access globally available services in which the scope is controlled by partitioning administrative domains (WS, SCA). In particular, most OSGi based applications make the assumptions that the platform contains a single application [3], and that there are not too many services. In this case, and despite the lack of any encapsulation mechanism, the application is isolated from other applications (being alone!) and runs only its components (being the only ones explicitly deployed before execution). These applications mostly take advantage of the dynamic deployment and update facility, which explains why OSGi has been adopted by embedded systems and by big applications like application servers (EJB), data bases (Oracle), IDE (Eclipse), for which the "mono application hypothesis" holds.

Unfortunately the simplicity and mono-application assumptions will not remain valid for long; first because complexity and scale increases; and second because a platform will have to support multiple applications. For example, at home, the set-top-box currently limited to TV decoding, will be in charge of "discovering" all the devices in the house, and to support all the applications that, potentially, will make use of these devices. It will not be possible to have a different platform for each one of these applications because they will have to share the same devices, and only a single authority can arbitrate the access and sharing (not talking about the cost and energy consumption of multiple platforms).

Ubiquitous computing, to a large extent, is related to the management of sensors and actioners whose number, nature, location and availability is unknown statically, and is changing over time (e.g. mobile devices). Such an application domain requires a platform in which devices and components can appear and vanish; be substituted or updated during the application's execution; such a platform is called a dynamic platform and OSGi is today the de-facto standard (low-level) dynamic platform. In the near future, OSGi will have to scale from a light-weight mono-application gateway to a full-fledged dynamic platform.

This paper presents Apam (APlication Abstract Machine), which is a layer on top of a standard OSGi implementation that provides the functionalities required for a general and full-fledged dynamic platform. More specifically, this paper addresses the topic of managing multiple applications in a service platform, using composite services. The paper presents the Apam concept of composite service and shows how this concept allows a service platform to manage multiple applications.

II. ENCAPSULATION MECHANISM

If the concept of composite service is defined to satisfy the hierarchical encapsulation principle, it should have at least the following properties:

- It "contains" other services (atomic or not),
- It hides the services it contains,

Before defining what a composite service should be, let us see first what a service is. In OSGi, a service definition is the tuple *serviceDefinition*==<*interface, properties*> and a service is defined as a tuple published in the "registry": *service*==<*serviceDefinition, address*>, with *interface* being a Java interface, *properties*=={<*name, value*>} describes the non-functional properties of that service, and *address* is the address of a Java object, instance of a class implementing (in the Java sense) the *interface*. Therefore, by nature, a service platform enforces a very strong information hiding mechanism: a service only shows what is published in the registry. Most notably, the code implementing the service (that we will call an implementation) is not visible. Indeed, it is the fundamental principle of a dynamic service platform: a service client, at run-time asks for a service, not for an implementation. A service request is the tuple *serviceRequest*==<*interface, logicalExpression*>, which means that the client wants a service that publishes that *interface*, and which properties satisfy the *logicalExpression*. In OSGi the logical expression is a first order logical expression over properties in LDAP syntax.

OSGi supports very few concepts, essentially bundles and services. A service is defined as <*interface, properties, address*>, and the concept of service implementation is simply undefined. With this definition, the concept of "a service contained in a service" barely makes sense.

We have to be more precise, and we introduce the concepts of service implementation and service instance. Let us define the concept of service implementation (implementation for short):*implementation*==<*class, {serviceDefinition}, {serviceRequest}*>, with *serviceDefinition* the set of services provided by this implementation, *class* a class implementing (in the Java sense) all the interfaces of the provided services, and *serviceRequest* the set of services that may be needed during execution of that implementation; when the *implementation* asks for a service[1], this service must have been registered and it must satisfy the request.

A composite implementation is defined as: *compositeImplementation*==<*mainImplem, {serviceDefinition}, {serviceRequest}*> where *class* is replaced by *mainImplem*. The main implementation (*mainImplem*) must provide (at least) all the composite *serviceDefinition.* By definition, the main implementation is contained in the composite. The services required by an implementation contained in the composite but not declared in the composite *serviceRequest,* will be considered as contained in that composite. A composite implementation is

[1] More precisely, when the thread that is executing an instance of the implementation class is asking the platform to get a service.

itself an implementation (its *class* being the *class* of its main implementation), and therefore it can contain other composite implementations. The first property of composites is satisfied (a composite service is a service containing other services). Note that this way of defining a composite is different from most composite definitions because it does not define, before execution the list of the implementations it will contain. Only the main implementation is statically defined, all other implementations will be selected dynamically; the composite definition gives the criteria to decide, at run time, what is inside, and what is outside the composite. This property is fundamental in a dynamic platform, since the services available at execution cannot be known statically.

The second property (a composite hides its content) is not a "natural" property in a service platform because the "registry" is a flat structure containing all the services available at any point in time; all services, contained or not in a composite, are visible and usable by anyone. We cannot use the OSGi registry, we developed a registry which "understand" our composite concept.

To the Apam registry, the composite constitutes a visibility scope. For example, suppose that an implementation IY, pertaining to a composite CY asks for a service Z. This request will be executed inside CY: it will be successful only if a Z implementation exists in CY or if it is possible to deploy one inside of CY. In the other cases, IZ is non existant or not *visible*, the service request fails. If Z pertains to the CY {serviceRequest}, the same algorithm is performed, not in CY but in the scope of the composite that contains CY. With this algorithm, an implementation belong and therefore is contained in all the composites that deployed it; the platform may hold different logical copies of the same implementation; but the composite still "owns" its implementations; it can delete, update or substitute them without any impact on the other copies and the other composites. A third party composite can be executed on our platform without being "messed up" with the other composites already running on that platform, even if they use the same implementations (with the same or different versions).

A composite implementation is an implementation in all aspects, and as such it can be instantiated, leading to a composite instance. A composite instance is an instance containing other instances, and at least one instance of the main implementation.

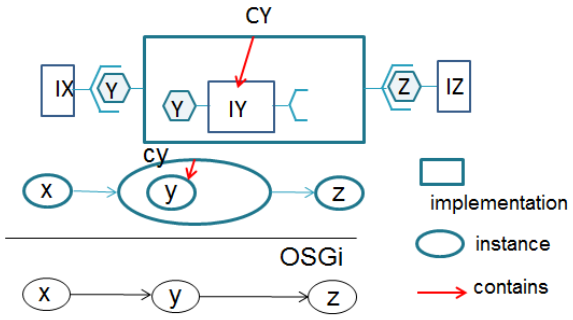

Figure 1: composites implementations and instances

In figure 1, it is supposed that CY {serviceRequest} includes Z, therefore implementation IZ is external to the composite. CY *delegates* to IY its Y interface, and *promotes* its implementation that requires Z.

III. Visibility Control, Protection and Sharing

In traditional platforms, each application (composite) is autonomous and runs in its own isolated space. In a service platform it is the opposite: all applications run in the same space and freely share everything. Autonomy and isolation are part of the best practices and should be applied whenever possible; but in a service platform, sharing is often desirable, and in ubiquitous computing, sharing is often required (e.g., sharing sensors). Therefore, a multi-application dynamic platform should allow designers to select and control the relevant strategy between, and including, these two extremes.

In Apam, with respect to the platform, a composite can be a lender and/or a borrower, or none. A composite is a **lender** if it allows other applications to use the elements it owns. A composite is a **borrower** if it prefers using an existing element (pertaining to another composite) instead of creating its own one. The Apam visibility control relies on the way resolution is performed.

When a client instance performs a *serviceRequest,* Apam does its best to return a service provider (an instance) satisfying the request, i.e., providing the required resource and satisfying the constraints. To do so, Apam performs as follows.

- A visible instance satisfies the request; it is returned. Otherwise,
- A visible implementation satisfies the request; an instance of that implementation is created and returned. Otherwise,
- An implementation satisfying the request is found in a visible repository; that implementation is deployed, an instance is created and returned. Otherwise,
- The resolution fails.

The issue is therefore to define what "visible" means.

A. Instance Visibility

A client instance pertaining to a composite instance *cc* can see an instance *inst* pertaining to a composite *cp* if

- *inst* pertains to *cc (cc = cp)* or
- *cp* lends *inst* to its friends, and *cp* is a friend, or
- *cp* lends *inst* to the application, and *cc* and *cp* pertain to the same application,
- *cp* lends *inst* to the whole platform.

cp is a friend of *cc* if a *friend* relationship is established from *cc* to *cp*. An instance pertaining to a single composite instance, the instances in a platform are organized as a forest. An application is defined as a tree in that forest (i.e., a root composite instance). Therefore, *cc* and *cp* pertain to the same application means they pertain to the same instance tree.

A composite can define which instances can be lent to other composite instances using the predefined attributes *localInstance*, *friendInstance* and *applicationInstance*. The

value of these attributes is an expression to be applied to instance properties. An instance cannot be lent if it matches the *localInstance* expression; it can be lent to *friend* composite instances if it matches the *friendInstance* expression; it can be lent to any composite of the same application if it matches the *applicationInstance* expression; and finally it is lent to the whole platform if it matches none. If it matches more than one expression, the most restrictive one is assumed.

Symmetrically, a composite designer must be able to decide whether or not to borrow the instances lent by other composites. For this purpose, he can specify the property *borrowInstance*=<*expression*>. If the requested resource matches the *expression,* the platform must try to borrow an instance if it exists. If the expression is not matched, an instance must be created. By default, the expression is "true", i.e., by default everything is shared.

B. Implementation Visibility

If no satisfactory instance is available, the platform tries to find an implementation from which it is possible to create an instance satisfying the request. In Apam, instances inherit the properties of their implementations, and implementations explicitly declare the properties specific to its instance. Therefore, from a *serviceRequest* Apam can compute the corresponding *implementationRequest*, i.e. a request with the same resource and with constraints that apply to implementations.

Therefore, a client instance pertaining to a composite implementation *CC* can see an implementation I pertaining to a composite implementation *CP* if

- *I* pertains to *CC* (*CC* = *CP*) or
- *CP* lends *I* to its friends, and *CP* is a friend, or
- *CP* lends *I* to the whole platform.

CP is a friend of *CC* if a relationship *friend* is established from *CC* to *CP*. A composite implementation can define which implementations can be lent to other composite implementations using the attributes *localImplem* and *friendImplem.* The value of these attributes is an expression. An implementation cannot be lent if it matches the *localImplem* expression; it can be lent to *friend* composite if it matches the *friendImplem* expression; it is lent to the whole platform by default. If it matches both expressions, it is supposed to be local.

If no satisfactory implementation has been found, the last trial is to find, among the available repositories, an implementation matching the *implementationRequest.* To that end we have extended the OSGi bundle repository with the implementation properties; this information is automatically computed during build by a Maven plug-in. Therefore, each repository is managed as an Apam registry. If a satisfactory implementation is found, it is dynamically deployed (in *CC*), and instantiated (in *cc*).

Symmetrically, an application designer must be able to decide whether or not to borrow the implementations lent by other composites. For this purpose, it can specify the property *borrowImplem*=<*expression*>. If the requested resource matches the *expression,* the platform must try to borrow an implementation. If not matched, or not existing, it must be deployed. By default, the expression is "true".

IV. Multiple Application Control

The challenge of a multi-application dynamic platform is to continue satisfying Software Engineering's best practices (which call for full isolation), while in a dynamic environment (which requires sharing); an apparent contradiction. Any non-trivial application is made of a number of clearly identified parts. Each application and composite designer must identify which parts must be private and hidden, which parts can be lent to others; and which parts can be borrowed. In Apam each "part" is modelled as a composite (implementation and instance), with its lender/borrower characteristics.

To illustrate, let us suppose the following scenario. In a house we have different kinds of display screens (e.g., TV, tablet, PC, smart phone), different audio renderers (e.g., TV, hifi) and two running applications: a media centre (*MC*) and an energy controller (*EC*).

The *EC* application contains its main implementation *mainEC* and requires a *display* service for configuring the application; *EC* distribution contains implementation *ECDisplay* which provides the *display* service on the tiny display embedded in the heater.

Media Centre (*MC*) is an application made of a number of implementations including composites *MCAudio* and *MCVideo*. *MC* holds the business part of the application (play lists, libraries, players, and so on) and requires an audio and a video device. *MCAudio* provides the video interface; it dynamically "discovers" the video devices and redirects the video streams it receives to the "best" device.

In this example, *EC* is a black box. *EC* wants a display; if possible the one available in the current context; if none are available, *ECDisplay* should be used. This strategy can be expressed in Apam as follows:

```
compositeImplementation EC2
   repository = http://.... ;
   mainImplem=mainEC ;
   //EC is a black box; but display can be borrowed
   borrowImplem=(interface=display);
   borrowInstance=false;
   localImplem=true; localInstance=true;
```

The *repository* attribute expresses where the implementations should be deployed from if needed. The only implementation that can be borrowed is the one implementing *display (*`borrowImplem = (interface = display))`. If a display exists in the platform it will be used (*MCVideo*), otherwise the display implementation found in the repository (*ECDisplay*) will be deployed and will be private to *EC*. All other implementations must either already be inside *EC*, or must be deployed from the *repository*.

[2] Currently, composite descriptions are in XML and expressions are in LDAP syntax; the syntax used here is for readability only.

No instance can be borrowed (`borrowInstance=false`); the instances required by *EC* must all be created inside *EC*. *MC* can be defined as follows:

```
compositeImplementation MC
   repository = http://.... ;
   mainImplem=mainMC ;
   borrowInstance=false; borrowImplem=false;
   localImplem=true ;
   //only audio and video are visible by all
   localInstance=(name!=audio && name!=video) ;
```

With this description, all implementations used by *MC* must pertain to *MC* (`borrowImplem=false`) and are not lent (`localImplem=true`); all instances must pertain to *MC* (`borrowInstance=false`) but the instances providing the *audio* and *video* services (which are also composites) are visible and usable by all.

```
compositeImplementation audio
   dynamicBind= audio;
   repository = http://.... ;
   mainImplem=mainAudioMng ;
   substitute=audio;
   borrowInstance=true; borrowImplem =true;
```

No constraints are set by the audio composite on the visibility of its devices. It means that the different devices are visible by all. Audio is a white box that allows its users to directly address its components (here the audio devices).

The *dynamicBind* primitive expresses that audio devices that appear / disappear dynamically must be automatically connected / removed to/from the audio composite; *substitute* means that an audio device that is used and that disappears must be substituted dynamically by another one and the audio flow redirected to the new device. The details of the dynamic behavior are not the topic of this paper and will not be described any further.

V. THE APAM PLATFORM

Apam differs from usual service platforms on different aspects. First, it clearly distinguishes between service specification, service implementation, and service instances. They are all first class objects, they all can be described, packaged, stored in repositories, selected, deployed, instantiated, and so on.

Second, Apam introduces the concept of composite, again clearly distinguishing composite implementations from composite instances. Composites are first class objects too; they can be described, packaged, stored in repositories, selected, deployed, instantiated and so on. In Apam, the word ***component*** is generic and applies to any one of the above concepts, be it, specifications, implementations or instances, either atomic or composite.

Third, the platform is in charge of not only managing existing services, but also of instantiating and deploying components when needed.

Fourth, Apam reifies all these concepts and their relationships into an Application State Model (ASM) causally connected to the underlying OSGi platform(s). The ASM represents a high level fully reflexive view of the current state of the system under execution.

The first need of composites is the support of the traditional principles of encapsulation and hierarchical decomposition which are well established in Software Engineering. To that end, a composite must be of the same nature of the objects it contains. Indeed, in Apam, the three basic entities (specification, implementation and instance) are each extended (in the Java sense) by their associated composite. Therefore, any operation that can be performed on an atomic concept can be performed on its composite. For example, a composite implementation can be deployed and instantiated exactly in the same way as atomic implementations. For its users, atomic and composite are identical, and in general do not need to be distinguished.

From a technical point of view, components (atomic and composite) are described (as XML files) in the development environment. Building a component [3] generates a bundle containing, as meta-information, the associated description. It means that when a component is deployed in OSGi, Apam is notified and reads the associated description. Therefore Apam contains a model of the current state of the system (ASM) but also an Application Component Model (ACM) containing the descriptions associated with the deployed components (whether currently used of not, instantiated or not). The ACM expresses both the consistency constraints the ASM must satisfy and the management strategies and rules governing the system's evolution. These rules include the visibility and protection as exemplified in this paper, but also the dynamic rules (see the audio composite above), and other rules. The ACM is reflexive: it is possible at run-time to observe the component model and to change the component and composite description, making rules fully dynamic too. When the ACM is changed dynamically, the ASM's consistency is checked (errors, if any, are notified but not fixed), and the new rules and strategies are immediately effective. This is an unusual property that can be used for the development of "meta rules" governing, for example, an autonomic system. This is an ongoing research activity, not presented here.

An important property of composites is that they factor out properties and strategies common to the whole composite. In this paper, out of visibility and protection, we sketched the deployment strategy (modeled as the *repository* attribute) and the dynamic strategy (modeled as the *dynamicBind* and *substitute* attributes), but other strategies are currently defined (distribution, recovery, autonomy), and others can be added. The system is extensible; each class of strategy is modeled by an independent model, interpreted by a specific manager. A manager is an Apam plugin, and Apam has a protocol for managers to synchronize and cooperate if needed. This is not described in this paper.

[3] We have developed a Maven plugin that performs these actions transparently. A bundle can contain many components; a "specification component contains the interface classes and the description; a "pure composite" bundle only constrains the meta-information, but can still be stored in repositories and deployed, as any other component.

In Apam, an application is simply a high level composite; an application can be used as a component of another higher level application. The platform itself is modeled as the root composite having as components all the first level applications deployed and/or running in the platform. Therefore, each composite (application) can define its isolation/collaboration strategy, but also the platform (the root composite) can define the "by default" isolation/collaboration strategy to be satisfied in this platform.

The Apam system is developed on top of iPOJO and extends the iPOJO system [4][5] (which itself extends OSGi [1]). Components are built under Eclipse using Maven; the Apam maven plugin injects code into the implementation (Java) classes. It is the injected code that calls Apam when a dependency needs to be resolved. Following the POJO (Plain Old Java Object) approach, the source code does not contain any information related to dynamicity, protection, structure and so on; implementations only contain the business code.

The experimentations so far have shown that the Apam system is both very efficient and flexible. The overhead when calling a method in another component (run-time performance) is the same as iPOJO (1.8.0 on Felix 3.2.2), but about 100 times faster than SCA [6] (Tuscany 1.3.2). The memory overhead is about 10% more than iPOJO, but 50% less than Tuscany. A more detailed account of performance comparisons is available in [7].

The current work is twofold: make Apam core robust enough to be used as an open research platform. Apam is extensible; research in under way to develop other managers (other non-functional aspects) and to experiment in the domain of pervasive and autonomic computing.

Documentation for Apam is available at http://wikiadele.imag.fr/index.php/Apam; the product will be available soon in the Lig forge.

VI. Related Works

Our approach builds on the many works on component models [8][9] related with encapsulation and hierarchical decomposition. We are particularly interested in the new requirements brought by dynamic and ubiquitous applications.

Work on dynamic component models has concentrated on the run-time reconfiguration of the application architecture [10][11], through the use of a reflective component runtime [12][13]. Our work on the Apam runtime platform pertains to this line of work, although this is not the focus of this paper.

Less attention has however been paid to the new requirements in terms of isolation, protection and visibility for multi-application dynamic platforms. Most dynamic component models use a strict black box approach for run-time composites like Sofa [14], or iPOJO composites [15] with the exception of Fractal [16] which allows for global shared instances.

Other approaches, like SCA [6] have tried to mix the dynamic capabilities of service platforms with the hierarchical decomposition of component models. However they propose to use a static black-box assembly of components at the lower granularity level, and a global scope for registered services.

Our work is also inspired by management of scope and visibility in programming languages [17], particularly its use to enforce isolation [18].

VII. Conclusion

Apam has been designed and implemented as a high level dynamic service platform in which application designers can structure their applications expressing to which extent each part (composite) must be isolated, must contribute to the platform (lending its components), must borrow platform components, or any mixture thereof. These features rely on the composite concept (implementation and instance). The main duty of composites is to factor out the properties and the management strategies to be applied on the components it contains.

Among the properties and strategies that can be associated with composites, the two major ones discussed in this paper are (1) structuring applications and (2) managing the level of isolation/collaboration between applications running simultaneously on the same platform.

The first need calls for the support of the traditional encapsulation and hierarchical decomposition principles. An Apam composite being a special case of components, recursive decomposition is "natural". Indeed, in most cases, there is no need to make the difference between atomic and composite components and the Apam resolution mechanism returns indifferently one or the other. However, traditional hierarchical decomposition also imposes a "black-box" approach, i.e., components contained in a composite are neither visible nor shared by components pertaining to other composites. This strict hierarchical decomposition leads to a partitioning that prohibits collaboration and sharing which is contrary to the needs of a dynamic platform that manages devices and services potentially used and shared by all the applications running of that platform..

The second need, which is managing the level of isolation/collaboration between applications, requires flexible mechanisms. Unfortunately, the usual service platform strategy where everything is visible and shared cannot be satisfactory when more than one application is running. To that end, Apam composites can express two classes of properties.

Lending components to the platform. A composite can express to which extent "its" components can be used by other composites. Apam provides 3 levels of visibility for both implementations and instances: default (visible by all), friend (only visible by composites with a friend relationship), and local: not visible at all. Instances have a 4th level of visibility: application (the instance is visible in the same instance tree).

Borrowing components from the platform. A composite can express to which extent it can borrow (use) visible components pertaining to other composites running on the platform. This is expressed by the *borrow* attribute. This property is also called **opportunism** since it consists in using

those services already running and available during execution, instead of installing and instantiating those planned before execution. Opportunistic strategies may be required to improve efficiency, to support collaboration, to avoid conflicts or simply because the application cannot instantiate the needed service (e.g., devices).

These attributes allow composites to be very fine grained in expressing their strategy, even without the full knowledge of its future components. With these features, a composite can use the full range of isolation/contribution.

On one extreme, "black-box", composites are defined with the attributes *localImplementation*, *localInstance*, set to *true* (i.e. no contribution) and *borrowImpementation* and *borrowInstance* set to *false* (i.e. no opportunism).

On the other extreme, the "scrambled eggs", Apam allows a complete mixture of implementations and instances as found in service platforms. It is the default in Apam; when running only legacy service applications, the platform contains only the root composite (the platform itself) and the attributes above with their default value. Therefore, iPOJO and OSGi legacy service-based applications run as usual; they only have to be rebuilt with our plugin. Any intermediate situation can be defined creating the "right composites" and setting these attributes with the "right expression".

A major issue with opportunistic applications is that they are dependent on the platform's context, and (usually) crash or freeze when required services are missing. In Apam, even opportunistic composites can provide the repositories containing its default distribution. If the required service is found it is used (opportunism); if missing the platform will deploy, install and instantiate the one found in the provided repositories. It allows applications to be collaborative, dynamic and opportunistic, without compromising availability and increasing reliability and resilience.

Conversely, well defined composites can increase their consistency and resilience, still executing in a dynamic platform with sharing and opportunism. Consistency can be increased because opportunism allows sharing the right service instead of deploying another one that can conflict with the one in place; resilience can be increased, relying on dynamic substitution toward alternative services unknown at development time.

The Apam platform has been developed and experimented. Even if preliminary, the experimentations are satisfactory enough to be confident that the challenge identified in the introduction can be met: getting both the software engineering best practices and the dynamic and opportunistic facilities in a multi-application dynamic platform.

REFERENCES

[1] OSGi Alliance, "OSGi Service Platform Core Specification Release 4", http://www.osgi.org, August 2005.

[2] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges", IEEE Computer, November 2007, pp. 38-45.

[3] P. Kriens, "Nested frameworks", http://www.osgi.org/blog/2010/01/nested-frameworks.html, 2010

[4] P. Lalanda and J. Bourcier, "Towards autonomic residential gateways", IEEE International Conference on Pervasive Services, 2006, pp 329-332.

[5] Apache Felix iPojo, http://felix.apache.org/site/apache-felix-ipojo.html

[6] OASIS, "Service Component Architecture Assembly Model Specification version 1.1" http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.pdf

[7] J. Estublier, G. Vega. Reconciling Components and Services. The Apam Component-Service platform . Submitted to SCC 2012

[8] I. Crnkovic, S. Sentilles, A. Vulgarakis and M.R.V. Chaudron, "A Classification Framework for Software Component Models", IEEE Transactions on Software Engineering, Vol 37, No. 5, September 2011.

[9] K. Lau and Z. Wang, "Software Component Models", IEEE Transaction on Software Engineering, Vol. 33, No. 10, October 2007.

[10] J. Magee and J. Kramer, "Dynamic structure in software architectures", Proceedings of the 4th symposium in Foundations of Software Engineering. 1996

[11] P. Oreizy, N. Medvidovic, R. Taylor, "Architecture-Based Runtime Software Evolution", Proceedings of the 20th International Conference on Software Engineering (ICSE'98), 1998.

[12] J.C. Georgas, A. van der Hoek and R. Taylor,"Using Architectural Models to Manage and Visualize Runtime Adaptation", IEEE Computer, Vol 42 No. 10, October 2009.

[13] T. Batista, A. Joolia and G. Coulson, "Managing Dynamic Reconfiguration in Component-Based Systems", Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005), 2005

[14] T. Bures, P. Hnetynka and F. Plasil,"SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model", Proceedings of the 4th International Conference on Software Enginering Research, Managament and Applications, 2006.

[15] C. Escoffier, R. S. Hall and P. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework", IEEE Int. Conference on Services Computing, USA, July 2007

[16] E. Bruneton, T. Coupaye and J-B. Stefani, "Recursive and Dynamic Software Composition with Sharing", Proceedings of 7th International Workshop on Component-Oriented Programming (WCOP 2002), 2002.

[17] P. H. Fröhlich and M. Franz, "On Certain Basic Properties of Component-Oriented Programming Languages", in Proceedings of the 1st Workshop on Language Mechanisms for Programming Software Components, October 2001.

[18] Ph. Fong and S. Orr, "Isolating untrusted software extensions by custom scoping rules", Journal of Computer Languages, Systems and Structures, Vol 36 No. 3 October 2010.