Apam Short Specifications

Motivations

We say a service is dynamic if it can appear and disappear, without notice, during the execution. We call dynamic application an application made, at least partially, of dynamic services.

Apam is an extensible service platform dedicated to the support of concurrent dynamic applications.

Apam does its best to simplify the design, development and execution of applications that must provide a "constant" service, in a moving and unpredictable context and in the presence of other, potentially competing, applications.

Being extensible, the functionalities and services provided by Apam depend on the available managers (a manager is an Apam plugin). However can be distinguished Apam core, which correspond to the minimum level of functionalities, extended Apam which is the standard delivery, and the specialized and third parties extensions.

Apam is an extension and generalization of iPOJO[1], which sees an application as a network of connected services. With respect to iPOJO, Apam adds (among other things) instantiation and deployment, and provides a more powerful composite concept and resolution mechanism. Apam has similarities with SCA, in particular its composites, but Apam is multi-applications and intrinsically dynamic (SCA is not) and offers a wider range of composite behavior.

Apam core.

Apam core goal is to:

- Define, control and adapt the application architecture in the presence of dynamic services.
- Define and enforce the protection, isolation and collaboration between concurrent, collaborating and/or competing applications.

To do so, Apam defines a component model (see the metamodel bellow); a component description contains the (meta) information needed for the compiler to produce the "right" executable components (bundles), and for the runtime to enforce the "right" behavior. At run time, a "State Model" represents the current state of the platform in terms of component instances and their relationships (wires and others), causally related to the underlying service platforms.

Roughly speaking, Apam core automates the creation of "wire" between services. This automatic wire management (creation, destruction, substitution) is performed in a moving context (dynamic services, devices discovery etc.) in conformance with the application definition and requirements. It leads to the automatic building of flexible and adaptable architectures.

[1] Apam is implemented on top of iPOJO, and compatible with iPOJO features.

The second point is based on the concept of composite, which is an envelope that expresses the contextual behavior of the components it contains, and its relationships with the other composites.

State Model and API.
Apam core provides a "state model" (ASM for Apam State Model) which is causally connected with one or more service platforms. The ASM is a fully reflexive model in terms of the entities defined in the metamodel bellow. Its API allows navigating and changing the ASM, as well as navigating and changing the definitions and strategies. The ASM sends events when its state is modified, whether by the API or as a consequence of the underlying service platform activity (service discovery, deployments, connections and so on).

Dependency management and the resolution process.
The goal of Apam is to simplify the writing of dynamic applications. To that end, the core functionality of Apam is the resolution of dependencies. A dependency "d" of a service (client) "C" is the definition of the resources (interfaces, messages …) that service C may need during its execution.
Resolving the dependency "d" of service "c" means selecting one or more service providers "P" that provide the resource(s) defined in "d" and satisfying the characteristics, constraints and preferences defined in "d". Once the providers selected, the resolution establishes a "wire" from the service instance "c" toward each one of the selected providers P. Since different strategies can be defined to resolve a dependency, the core delegates the "dependency resolution" to a number of "dependency managers", which collectively are responsible to find out the "right" provider(s) as the resolution of the dependency.

From service to components: the composites.
The core defines a concept of composite which goal is to fill the full spectrum between the flat and unprotected structure, as found in usual service platforms (where any service can use and be used by any other one), and the black box structure, as found in many component models (where the content of a composite is fully hidden from outside).
Each composite can define its level of "transparency", and can define the strategies to be used inside the composite.
These mechanisms are primarily intended for the management of multiple applications or sub-systems sharing the same platform and devices.

Apam core managers

Three classes of managers are defined:

- dependency managers, called when a dependency needs to be resolved,
- Property managers, called when a property of a component is modified, and
- Dynamic managers, called when a service appears or disappears.

Apam extensibility is based on managers to be provided by third parties; however at least two managers are closely related to the core and must be provided: ApamMan and Dynaman. Of course, a default implementation of these managers is provided is the distribution.

ApamMan is the default dependency manager. ApamMan tries to resolve a dependency looking into the components currently running in the platform, taking into account the visibility expressed in the client composites. If an implementation is found but no instance is available, ApamMan creates an instance of the selected implementation.

OSGiMan is a dependency manager that tries to resolve a dependency looking in the OSGi registry of the current machine. It allows the integration in Apam of OSGi legacy services.

Dynaman is the default dynamic manager. Dynaman interprets the strategies defined in the dependencies when a resolution fails, and when a component appears or disappears.

With these two managers, all the information, properties and characteristics defined in the component definition are fully enforced. Another implementation of these two managers can significantly change the core semantics and is therefore discouraged.

Extended Apam.

The standard distribution comes with at least the following basic managers.

OBRMan is a dependency manager that extends Apam with dynamic deployment. During a resolution, OBRMan is called if ApamMan did not found a convenient service (the right service is not currently running in the platform, or is not visible). OBRMan looks in a number of local or remote bundle repositories to find out a service that satisfies the dependency requirements; if found, the corresponding bundle is deployed on the current machine. See Compilation and OBR repositories bellow for more details on how OBRMan works.

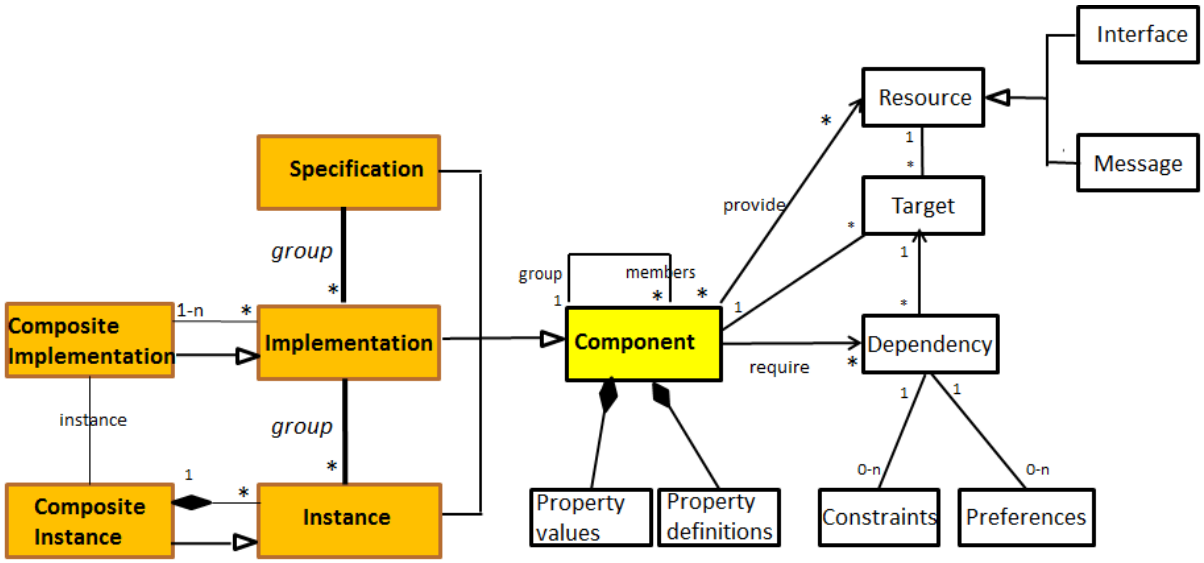
Distriman is a dependency manager that extends Apam with distribution. During a resolution, if ApamMan did not found a convenient service (the right service is not currently running in the platform, or is not visible), Distriman can be called (after or before or instead OBRMan). Distriman looks on the network to find out a visible Apam machine on which a convenient service is currently running. If found, a proxy is created from the current machine toward the distant service and the resolution returns the proxy address.
Note that, even without Distriman, remote services can be discovered (by ROSE) and used as if they were local; it is the case in particular for web services, and devices.

Specialized and domain specific managers.

ConflictMan is a dynamic manager specialized in the resolution of access conflict between two or more applications that require an access to the same exclusive services. This manager is intended to solve the conflicts toward shared exclusive devices. See Access conflict below.

Third parties are encouraged to develop specialized managers as a solution to their needs.

The APAM metamodel.



Apam is an advanced service platform, a component model, and a development framework. Apam components are typically developed under Eclipse with Maven as builder.
A single Eclipse project can host a number of Apam components; the metadata associated with the project must contain the declaration of all these components. For project S2Impl, the associated metadata is typically in the repository $project/src/main/resources/Metadada.xml, or it is indicated in the .pom as well as the Maven plug-in required to compile and build Apam components:

```
<plugin>
   <groupId>fr.imag.adele.apam</groupId>
   <artifactId>ApamMavenPlugin</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <executions>
      <execution>
         <goals>
              <goal>ipojo-bundle</goal>
         </goals>
         <configuration>
            <metadata>src/main/resources/S2Impl.xml</metadata>
         </configuration>
      </execution>
   </executions>
</plugin>
```

An Apam metadata file is an xml file that should start with the following header:

```
<apam xmlns="fr.imag.adele.apam"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="fr.imag.adele.apam http://repository-
apam.forge.cloudbees.com/release/schema/ApamCore.xsd" >
```

The xml examples below are supposed to be found in such an Apam metadata file.

Components

Apam is based on the concept of component. Components can be of three types: Specification, Implementations and Instances, that share most of their characteristics.

The group concept.
As shown in the metamodel, components are related by a "group-members" relationship. A specification is a group whose members are implementations, and an implementation is a group whose members are instances. A group-member relationship establishes a de facto inheritance between the group and its members. More precisely, all the characteristics of the group (its properties, its provided and requires resources) are automatically inherited by all its members, like in a class-instance relationship.

A specification is a first class object that defines a set of provided and required resources (in the java sense). Complete compositions can be designed and developed only in term of specifications.

```
<specification name="S2" interfaces="apam.test.S2, apam.test.AB"
      messages="apam.test.M1, apam.Test.M2" >
      …..
</specification>
```

The example shows how are declared specifications. Specification S2 provides two interfaces, *apam.test.S2* and *apam.test.AB* and produces two messages of type *apam.test.M1* and *apam.Test.M2.* Required resources will be discussed later.

An implementation is related by an "implements" relationship with one and only one specification. An implementation is an executable entity (in Java) that implements all the resources defined by its associated specification, and that requires at least the resources required by its associated specification. In practice, an implementation must define a class that implements (in the java sense) the interfaces of its specification.

```
<implementation name="S2Impl" specification="S2"
      classname=" apam.test.S2Impl"
      push="produceM1, produceM2, produceM3"
      interfaces="apam.test.AC" >
</implementation>
```

In this example, the implementation *S2Impl* implements specification *S2* and therefore provides the same interfaces (*apam.test.S2* and *apam.test.AB*) and messages *apam.test.M1* and *apam.Test.M2* (provided by methods *produceM1*, *produceM2*) as defined by *S2*. Additionally it also provides interface *apam.test.AC* and the messages produced by method *produceM3*. Messages will be discussed later.

An instance is related by an "instanceOf" relationship with one and only one implementation. An instance is a run-time entity, represented in the run-time platform (OSGi) as a set of Java objects, one of which is an instance (in the Java sense) of its associated main class implementation. In the underlying service platform, an instance can be seen as a set of services, one for each of the associated specification resource; in Apam it is an object.

Instances are essentially created automatically at run-time, but they can also be declared, as follows:

```
<instance name="InstS2Impl" implementation="S2Impl" >
      <property name="XY" value="false" >
      ….
</instance>
```

When the bundle containing this declaration will be loaded, an instance called *InstS2Impl* of implementation *S2Impl* will be created with the properties indicated (here XY=false).

Component life cycle

During execution, in Apam, a component has a single state: it is either existing (and therefore available and active), or non-existing.

Property management

Properties are pairs (name, value), name is a string, and value is a typed singleton or set. Names and values are case sensitive.

Property definition

Properties are typed; the type is either a basic type, or a set of elements of a basic type. The definitions of properties are as follows:

```
<specification name=”S1” ….
      <!-- singleton values -->
      <definition name="hostName" type="string" />
      <definition name="speed" type="int" />
      <definition name="location" type="living, kitchen, bedroom"
            value="bedroom" />

      <!--set values -->
      <definition name="OS" type="{Linux, Windows, Android, IOS}" />
      <definition name="names" type="{string}" value=”tom, jacques”/>
      <definition name="notes" type="{int}" value=”1, 2, 5, 74”/>
      ……
</specification>
```

Basic types are “int”, “integer”, “string”, “boolean” or enumeration. An enumeration is a comma separated list of string. Values must not contain coma. White-spaces are ignored around the commas. A definition can include a default value, as for “location” above.
A type can define a set if inside braces. The value of the “OS” property can be a set of the enumerated values; “names” is a set of string while “notes” is a set of integers.

A definition in a component is used to set that property to its members. For example the definitions above can be instantiated on any implementation of specification “S1” or on instances of “S1” if the implementation did not instantiate the property, as for example, “hostname” because that property makes sense only on instances.
A property name cannot be one of the Apam reserved names[2].

[2] Final properties are: name, spec-name, impl-name, inst-name, composite, main-component, main-instance, interface, message, provide-interface, provide-message, provide-specification.

A property i.e. a pair (name, value) can be instantiated on a component C, if the property is defined in C, or declared in any group above of C and not instantiated, and if the value matches the property type.
Instantiation can be performed in the component definition as in the following example, by API when creating the component, or by API calling the method "C.setProperty (String name, Object value)".

```
<implementation name="S1Impl" classname="XY.java" specification="S1"
            shared="false">
      <property name="location" value="living" />
      <property name="OS" value="IOS, Android" />
      <property name="language" value="Java" type="Java, Python, C++" />
      <definition name="fieldAttr" field="myField" internal="true"
                  value="2"/>
      …
</implementation>
```

In this example, attributes *location* and *OS* are valid since defined in the associated specification, and attribute *language* is valid because it is defined and instantiated at the same level.
In this implementation, the attribute "*fieldAttr*" is associated with the field *myField* in the java source code of class "XY.java". By default (internal="false") the value of the attribute and the value of the java field are synchronized, both ways (it can be set either by the XY class, assigning a value to the variable, of by the Apam API using the method "setProperty ("fieldAttr", "aValue")". If internal="true", only the program can set the attribute value, but the attribute value is visible. If a value is indicated, it will be the initial value of the variable, even if internal.
Specification attributes must be both declared and instantiated at the specification level.

```
<specification name="S1" interfaces="…"  >
      <property name="S1-Enum" value="v1" type="v1, v2, v3"/>
      <property name="S1-Attr" value="Hello" type="string"/>
      …
</specification>
```

In this example, the only valid properties for S1 are S1-Enum and S1-Attr, and they are inherited by all S1 implementations and instances.

Property inheritance.
As for any characteristics, a component inherits the attributes instantiates on its group (and recursively). An inherited property cannot be set or changed; it is updated if it changes in the group.

Technical Domain properties
The technical domain (i.e. Specification, Implementation, Instances) defines a few properties which semantics has been defined by Apam core.
These properties can be associated with any component. If defined with the same syntax as domain specific properties they are the following:

- <definition name="shared" type ="boolean" value="true" />
  share="true" means that the associated instances can have more than one incoming wire. share= "false" means that each instance can have at most one incoming wire.

- <definition name=”singleton” type=”boolean” value=”false” />
  Singleton=”false” means that each implementation can have more than one instance.
  singleton=”true” means that the implementation can have at most one instance.
- <definition name=”instanciable” type=”boolean” value=”false” />
  instanciable =”false” means that it is possible to create instances of that implementation.
  instanciable =”true” mean that it is not possible to create instances of that implementation (devices for instance).

These properties are indicated in the component tag:

```
<specification name="S2" singleton="false" instantiable="false"
      shared="false" interfaces="apam.test…."   >
```

For user convenience, these properties, as well as some final properties, are generated as domain specific attributes. It allows users to use these attributes in filters.

Callback method
Callback methods are called when a component instance is created, and when it is removed. They can be declared in the specification or in the implementation as follow:

```
<specification name="S1" interfaces="…">
   <callback onInit="start" onRemoved="stop" />
</specification>

<implementation name="S1Impl" classname="XY.java" specification="S1"
            shared="false">
   <callback onInit="start" onRemoved="stop" />
</implementation>
```

The Java program must contain methods `start` and `stop` (names are fully arbitrary):

```
          public void start () { }
or        public void start (Instance inst) { }
          public void stop () {}
```

The method declared as the "onInit" flag ("start" in the example) is called when an instance of the implementation is created (explicitly or if it “appears”); the method declared as the "onRemoved" flag ("stop" in the example) is called when the instance disappears.
The onInit method can have, as parameter, the actual Apam instance (`this == inst.getServiceObject()`).

Execution and OSGi bundle repositories (OBR)

At execution, Apam (more exactly managers like OBRMan), can deploy dynamically Apam components (more exactly the bundles containing these components) potentially from remote repositories. These managers receive their model each time a composite type is deployed, and should resolve the dependencies with respect to the current composite type model.
In the special case of ObrMan, the model associated with composite type “Compo” is found in the directory “${basedir}/src/main/resources/Compo.ObrMan.cfg”.

That file has the following syntax:

```
LocalMavenRepository = [true | false]
DefaultOSGiRepositories = [true | false]
Repositories=http:/……../repository.xml \
 File:/F:/…… \
 https:/…..
Composites=S1CompoMain CompoXY …
```

Attribute `LocalMavenRepository` is a Boolean meaning if yes or not, the local Maven repository, if existing, should be considered.
Attribute `DefaultOSGiRepositories` is a Boolean meaning if yes or not, the Obr repository mentioned in the OSGi configuration should be considered.
Attribute `Repositories` is a list, space separated, of OBR repository files to consider. The order of this list defines the priority of the repositories.
Attribute `Composites` is a list, space separated, of Apam composite types. It means that the repositories defined for that composite type should be considered. The order of this list defines the priority of composites repositories. These composites must be present in Apam at the time the composite is installed, they are ignored otherwise.

The list of repositories defined by this file is the list of repositories to associate with that composite type.

The order of the attributes in the file defines the priority in which the resolution will be done by the OBR, for example:

In this model:

```
LocalMavenRepository=true
DefaultOSGiRepositories=true
Repositories=http:/……../repository.xml
Composites=S1CompoMain
```

First, we will check the `LocalMavenRepository,` then `DefaultOSGiRepositories` then `Repositories` and finally `Composites.`

The default models associated with the Apam root composite type are found in the OSGi platform under directory "./conf/root.OBRMAN.cfg". If this file is missing, its content is assumed to be `LocalMavenRepository=true DefaultOSGiRepositories=true.` For composites types that do not indicate an OBRMAN.cfg model, ObrMan uses the root model.

Apam relies on the OBR mechanism for dynamically deploying the bundles containing the required packages. For that reason the Apam Maven plug-in adds in the OBR repository the dependency toward the Apam specifications, along with the right version.

Dependency management and resolution strategies

The traditional resource management strategy is to first gather all the resources needed by an application before starting it. Unfortunately, in our context, between time t0 at which a service s is started and time t1 at which it needs a service provider P, many things may occur. P may be non-existing at t0, but created before t1; P may be unavailable or used at t0 but released before t1; a provider of P (say p1) may be available at t0 but at t1 it is another provider (say p2) that is available. Therefore, each service (and applications) should get the resources it needs only when they are really needed. Conversely, resources must be released as soon as

possible because they may be needed by other services. It is the lazy strategy. Therefore Apam is fully lazy by default. However, an eager strategy can be imposed by the composite (see XXX).

We call **resolution** the process by which a client finds the service provider (an instance) it requires.

In Apam, a dependency is defined towards a component (specification, implementation or instance) or a resource (an interface or a message) defined by their name, constraints and preferences (see the metamodel above).
If the dependency is defined toward a component, the resolution consists first in finding that component and then to select one of its member satisfying the constraints and preferences, and recursively until to find the instance(s).
If the dependency is defined toward a resource, the resolution consists in finding a component providing that resource and satisfying the constraints and preferences, and recursively until to find the instance(s). If no instance satisfies the constraint but an implementation is available, an instance is created; otherwise the resolution fails.
The components are found either in the platform (the currently running services), or in a repository, local or distant (OBR, Maven, …). Since the component description is the same in all repositories, including the platform, the same constraints and preferences apply indifferently in all repositories. The available repositories are per composite, (see "Execution and OBR repositories" above). If found in a repository, the selected component is transparently deployed and instantiated; therefore, for the client developer, it makes no difference if the component is found in the machine or in any repository. Conceptually, all the components are in the machine (like between the virtual memories and the physical memory).

Nevertheless it is always possible for a resolution to fail i.e. no convenient implementation or instance can be found, in that case, by default, null is returned to the client i.e. the client code must check its variable before any use, which is relevant only if the dependency is optional. On all the other cases, the client would like to assume that its variable is always conveniently initialized. The strategy in this case is controlled by the "fail" property associated with dependencies. For example:

```
<dependency specification="S2" field="s2" id="fastS2"
      fail= "wait" | "exception" exception="fr.imag. ….failedException" />
```

Fail= "wait" means that if the resolution fails, the client current thread is halted. When a convenient provider appears, the client thread is resumed with its dependency resolved against that provider. Therefore, the client code can always rely on a satisfactory resolution, but may have to wait.
Fail ="exception" mean that, if the dependency fails, an exception is thrown, as defined in the exception tag. If no user exception is defined the Apam default *"ResolutionException"* is thrown. The source code is supposed to catch that exception.
Exception="Exception class" mean that, if the dependency fails, the associated exception is thrown. The Exception class must be exported in order for Apam to see the class (using the Admin), and to throw the exception.

If, for any reason (failure, disconnection, …) the instance used by a dependency disappears, Apam simple removes the wire, and a new resolution of that dependency will be intended at the next use of the associated variable. It means that dynamic substitution is the default behavior.

Dependency cardinality
A “simple” dependency is associated with a simple variable in the Java code. At any point in time, the variable points to zero or one provider.
A multiple dependency is associated with a variable that is a collection i.e. an “array”, a “Set”, a “Vector” or a “List”. Such a dependency therefore leads to a set of service providers. When the dependency is resolved for the first Apam, the dependency is associated with all the instances implementing the required resources, available at the time of resolution. If none are available, one is instantiated if possible, the resolution fails otherwise.

```
<dependency specification="S3Compile" id="S3Id" multiple=”true”>

<interface field="fieldS3" multiple=”true”/> <!– multiple is useless -->
```

The multiple attribute is very useful only for specification dependencies, since there is no other way, at that level, to know. For implementations, the field type (Collection or not) indicates if the dependency is multiple or not. If the field is a collection, the attribute multiple can be missing, it is assumed to be *true*, it can be set to *true*, but is cannot be *false*.

Once the dependency resolved, any new instance (of the right type) appearing in the system is automatically added to the set initially computed; similarly, each time an instance disappears, it is removed from the set of instances. This even can be captured in the program, if callbacks are indicated:

```
<dependency field="fieldT" added="newT" removed="removedT" />
```

In this example, if `fieldT` is a set of type `T`, the Java program must contain a method `newT` and `removedT` (names are fully arbitrary) :

```
          Set<T> fieldT ;

          public void newT (T t) { }
or        public void newT (Instance inst) { }
          public void removedT () {}
or        public void removedT (Instance inst) {}
```

The method `newT` must have as parameter either an object of type `T`, or an object of type Instance (`fr.imag.apam.Instance`). This method is called each time an object (of type `T`) is added in the set of references, this object is the parameter. Similarly, the method `removedT` is called each time an object is removed from the set; it may have the Apam instance object as parameter (warning: it is an isolated object without a real instance inst.getServiceObject()==null)
About messages, the `newM1` method is called each time a new provider is added in the set of the `M1` message providers, and `removedM1` is called when an `M1` provider is removed.

Complex dependencies
A complex dependency is such that different fields and messages are associated with the same provider instance. The provider must implement a specification, and the different fields must reference the different resources defined by that specification.

```
<dependency specification="S3Compile" id="S3Id">
      <interface field="fieldS3" />
```

```
        <message method="mes1" />
        <interface field="field2S3" />
    </dependency>
```

In the example, the dependency *S3Id* is a dependency toward one instance of the specification *S3Compile*. That instance is the target of fields *fieldS3* and *field2S3*, and the provider of message *mes1*. For dependencies with cardinality multiple, all variables are bound to the same set of service providers (internally, it is the same array of providers). It means that that dependency is resolved once (when the first field is accessed), and if it changes, it changes simultaneously for all fields.

Message

Following our metamodel, a component provides resources (interfaces or messages) and dependency can be defined against interfaces or messages. Therefore a component can be a message provider, or a message requester.

A message provider must indicate in its declaration header, as for interfaces, the type of the provided messages, and for implementations, the associated fields (see example above).

```
<specification name="S2" interfaces="apam.test.S2"
      messages="apam.test.M1, apam.Test.M2" >
      …..

<implementation name="S2Impl" specification="S2"
      push="producerM1, producerM2"
      interfaces="apam.test.AC"     ……
```

The S2Impl implementation should contain the **methods** producerM1 and producerM2:

```
      public M1 producerM1 (M1 m1) { return m1; }
```

Each time the producer calls the produceM1 method, Apam considers that a new M1 message is produced. There is no constraint on the method producerM1 parameters, but it must return an M1 object. A dependency can be defined against messages in a similar way as interfaces, but methods instead must be indicated in the case of push interactions or a java.util.Queue field in the case of pull interactions, as in the following examples.

```
<dependency pull="queueM1" />
<dependency field="fieldS2" />

<dependency specification="S2Compile" >
      <interface push="getAlsoM1" />
      <message pull="anotherQueueM1" />
</dependency>

<dependency push="gotM2" />
<dependency pull="queueM2" />
```

The first line is a simple declaration of a message dependency; analyzing the source code it is found that queue M1 is a field of the type java.util.Queue that has a message of type M1 as a paramterType and therefore is associated with the message M1 dependency. The associated Java program should contain:

```
Set<S2> fieldS2 ;
S2 anotherS2 ;

Queue<M1> queueM1;
Queue<M1> anotherQueueM1;
public void getAlsoM1 (M1 m1) { ....}

Queue<M2> queueM2;
public void gotM2 (M2 m2) { ..... }
```

The Queue are very special field: Queue are instantiated by ApAM at the first time call, then ApAM place all then new messages inside them. If there is no new M1 value available the queue is empty, and if there is no producer the Queue is null (see resolution policy)

At the first call to these queues, the corresponding M1producers are resolved and connected to the queue. If the dependency is multiple, all the valid M1 producer will be associated to the queue, otherwise a single producer is connected. In this case, as for usual dependencies, it is the client that has the initiative to get a new value. We call it the *pull* mode.

A producer my can also declare methods that return is a set of message:

```
public Set<M1> producerM1 (...) { ....}
```

When these methods are called, ApAM will consider that all the returned objects are provided messages.

For consumer, The declared method is void (push interactions), with a message type as parameter (M2 here), this method will be called by Apam each time a message of type M2 is available. In this case it is the message provider that has the initiative to call its client(s). The connection between client and provider is established at the first call by the provider to its produceM2 method. In the example, the method gotM2 will be call each time an M2 message is produced by one of the valid M2 producers.

In the previous examples, the raw data of type M1 and M2 is received by the clients. If more context is required, the injected methods or Queue can declare Message<M1> instead of M1; Message being a generic type defined in ApAM that contains an M1 values and information about the message: producer id, time stamp, and so on.

For multiple message dependencies, as for interfaces, it is possible to be aware of the “arrival” and “departure” of a message provider:

```
<dependency push="getM1" added="newM1Producer" removed="removedM1Producer"
/>
```

With the associated methods, as shown above for interfaces.

Constraints and preferences

```
<dependency specification="S3Compile" id="S3Id">
      <interface field="fieldS3" />
      <constraints>
            <implementation filter="(apam-composite=true)" />
```

```
            <instance filter="(&amp;(testEnum*&gt;v1,v2,v3)(x=6))" />
            <instance filter="(&amp;(A2=8)(MyBool=false))" />
      </constraints>
      <preferences>
            <implementation filter="(x=10)" />
            <instance filter="(MyBool=false)" />
      </preferences>
</dependency>

<definition name="testEnum" type="v1, v2, v3, v4, v5" value="v3" />
```

In the general case, many provider implementations and even more provider instances can be the target of a dependency; however it is likely that not all these providers fit the client requirements. Therefore, clients can set filters expressing their requirements on the dependency target to select. Two classes of filters are defined: constraints and preferences. Filters can be defined on implementations or instances.
Constraints on implementation are a set of LDAP expression that the selected implementations MUST ALL satisfy. An arbitrary number of implementation constraints can be defined; they are ANDed.
Similarly, constraints on instance are a set of LDAP expression that the selected instances MUST ALL satisfy. An arbitrary number of instance constraints can be defined; they are ANDed.

Despite the constraints, the resolution process can return more than one implementation, and more than one instance. If the dependency is multiple, all these instances are solutions. However, for a simple dependency, only one instance must be selected: which one ?
The preference clause gives a number of hints to find the “best” implementation and instance to select. The algorithm used for interpreting the preference clauses is as follows:
Suppose that the preference has n clauses, and the set of candidates contains m candidates. Suppose that the first preference selects m’ candidates (among the m). If m’ = 1, it is the selected candidate; if m’=0 the preference is ignored, otherwise repeat with the following preference and the m’ candidates. At the end, if more than one candidate remains, one of them is selected arbitrarily.

Contextual dependencies

A component (instance) is always located inside a composite (instance). The composite may have a global view of its components, on the context in which it executes, and on the real purpose of its components. Therefore, a composite can modify and refine the strategy defined by its components; and most notably the dynamic behavior.

For example, if composite *S1Compo* wants to adapt the dynamic behavior of all the dependency from its components and towards components the name of which matches the pattern "*A*-lib”*, it can define a generic dependency like :

```
<composite name="S1Compo" …
      …
   <contentMngt>
      <dependency specification="A*-lib" eager=“true” id=”genDep”
         hide=“true” | “false” exception=”….CompositeDependencyException”/>
```

Suppose a component “*S1X*” pertaining to S1Compo has defined the following dependency:

```
<dependency specification="Acomponent-lib" id="S1XDep" fail="exception"
      exception="….S1XDependencyException"/>
```

When an instance *inst* of *S1X* will try to resolve dependency *S1XDep*, since *Acomponent-lib* matches the pattern *A*-lib,* the generic dependency overrides the *S1X* dependency flags (*fail* and *exception*) and extends *S1XDep* with the *eager* and *hide* flags.

Eager="true" means that the *S1XDep* dependencies must be resolved as soon as an instance of *S1X* is created. By default, eager=false, and the dependencies is resolved at the first use of the associated variable in the code.

Exception="Exception class" means that, if the *S1XDep* dependency fails, Apam will throw the exception mentioned in *genDep* (the full name of its class) on the thread that was trying the resolution. This value overrides the exception value set on *S1XDep.*

Hide="true" means that, if the *S1XDep* dependency fails, all the *S1X* instances are deleted, and the *S1X* implementation is marked invisible as long as the dependency *S1XDep* cannot be resolved.
Invisible means that *S1X* will not be the solution of a resolution, and no new instance of *S1X* can be created. All *S1X* existing instances being deleted, the actual clients of *S1X* instances, at the next use of the dependency, will be resolved against another implementation and another instance. But if a thread was inside an instance *inst* of *S1X* at the time its dependency is removed, the thread continues its execution, until it leaves *inst* normally, or it makes an exception. No other thread can enter *inst* since it has been removed.

If the *hide* flag is set, it overrides the component *wait* flag because the instance will be deleted. But *hide* and *exception* are independent which means that in case of a failed resolution, the client component can both receive an exception and be hidden (but cannot wait if hidden). If there is no composite information, only the component "fail" policy applies. If both exceptions are defined, only the composite one is thrown.

This ensures that the current thread which is inside the instance to hide has to leave that instance, and that no thread can be blocked inside an invisible instance.

Important notes: The hide strategy produces a failure backward propagation. For example, if *S1XDep* fails, Apam hides component *S1X* and deletes all the *inst* incoming wires. If an instance *y* of component *Y* had a dependency toward *inst*, this dependency is now deleted. At the next use of the *y* deleted dependency, since *S1X* cannot longer be a solution (it is hidden), Apam will look for another component satisfying the *Y* dependency constraints. If this resolution fails (no other solution exist at that time), and if the *Y* dependency is also "hide", *y* is deleted and *Y* is hidden. The failure propagates backward until a component finds an alternative solution.
This had two consequences: first, it ensures that the application is capable to find alternative solutions not only locally but for complete branches (for which all the dependencies are in the hidden mode). Second, the components are fully unaware of the hidden strategy; the strategy is per composite, which means this is only contextual; it is an architect decision, not an implementer one.

Contextual constraints

Generic dependencies can express generic constraints:

```
<composite name="S1Compo" …
      …
<contentMngt>
 <dependency specification="A*-lib" …. >
         <constraints>
            <instance filter="(OS=Linux)" />
         </constraints>
     </specification>
   </contentMngt>
```

In the example, all the components trying to resolve a dependency toward instances of specifications matching `A*-lib` will have the associated properties and constraints.
The constraints that are indicated are **added** to the set of constraint, and appended to the list of preferences, for all the resolutions involving the matching components as target.
In the example, all instances of specifications matching `"A*-lib"` must match the constraint OS=Linux. Note that it is not possible to check statically the constraint, since the exact target specification is unknown, and therefore we do not know which properties are defined. If a property, in a filter, is undefined, the filter is ignored. For example, if an instance does not have the "OS" property, the filter containing the expression (OS=Linux) is ignored.

Visibility control

In Apam, with respect to the platform, a composite (implementation or instance) can export its components (implementations or instances), or import components exported by other composites. This control is performed during the dependency resolution. A dependency from an instance client c in composite cc toward a provider instance p of implementation P is valid (i.e. a wire will be created from c to p) if :

1. visible (c, p) ∧ import(cc, p)
2. visible (c, P) ∧ import(cc, P) ∧ instantiable(P).

The following provides the semantics of predicates visible (x, p) and import (cc, p).
The <*expression*> is either a Boolean ("true" or "false") or an LDAP filter to be applied to the component candidates.

Importing components.

A composite designer must be able to decide whether or not to import the instances exported by other composites. This is indicated by the tag <*import Implementation*=<*expression*> or *Instance*=<*expression*>. If the target implementation or instance matches the *expression*, the platform must try to import it if possible. By default, the expression is "true", i.e., the composite first tries to use whatever is available in the platform.

```
<import implementation="(b=xyz)" instance="false"/> <!—default is true -->
```

Import (cc, p) is true if, in composite cc, component p matches the corresponding expression (implementation if p is an implementation, instance otherwise).
In this example, the current composite cc will try to import the implementations that match the expression `(b=xyz)`, but never an instance (`instance="false")`.

If we have `<import implementation="false" instance="false"/>`, the composite will have to deploy all its own implementations from its own repositories, and create all its instances. It means that it is auto-contained and fully independent from the other composites and components. It can be safely (re)used in any application. Nevertheless, its resolution constraints can include contextual properties such that it can adapt itself to moving context, still being independent from its users.

Exporting components

Visible (x, y) is always true if x and y are in the same composite.
If no `export` tag is present, visible (x, y) is true.
If an export clause is present, only those components matching the export clause can be visible:

```
<export    implementation="Exp" instance="Exp"/> <!-- true by default -->
<exportApp instance= "Exp" />
```

`Export` means that the components contained in the current composite matching the expression are exported toward all the composites. An implementation can be inside more than one composite type with different export tags; the effective export if the most permissive one[3]. Export(x) is true by default.
For example `<export implementation="false" instance="false"/>` means that the composite is a black box which hides its content; it does not share any of its service with other composite (except if exportApp allows some services to be visible inside the current application).
`ExportApp` means that the *instances* contained in the current composite and matching the expression can be imported by any composite pertaining to the same application.
ExportApp(x) is false by default.
For example, `<export instance="false"/><exportApp instance="true"/>` means that the services the current composite instance contains are visible only inside the current application.
An instance pertains to a single composite instance; therefore 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, *two composite instances* pertain to the same application if they pertain to the same instance tree.
By default (none of the above tags are present) a composite exports everything it contains, and imports everything available.

In summary, visible (x, y) = true if one of the following expressions is true:

- composite(x) = composite(y) or
- export (y) = true or //true if no export tag
- (exportApp(y) = true) ∧ (app(x) = app(y)) //false if no exportApp tag

With composite(x) the composite that contains x; app(x) the application that contains instance x; export (x)=true if x matches the export expression, and exportApp(x) =true if x matches the exportApp expression.

[3] An implementation is inside a composite type only if it has been deployed by that composite type.

Promotion

A composite type is an implementation, and as such it can indicate its dependencies, as for example:

```
<composite name="S1Compo" mainImplem="S1Main" specification="S1" >
      <dependency specification="S2" multiple="true" id="S2Many">
            <constraints>
                  <implementation filter="(apam-composite=true)" />
                  <instance filter="(Scope=global)" />
            </constraints>
      </dependency>
      <dependency interface="fr.imag.adele.apam.test.s2.S2" id="S2Single">
            <preferences>
                  <implementation filter="(x&gt;=10)" />
            </preferences>
      </dependency>
```

This definition says that composite *S1Compo* has a dependency called *S2Many* towards instances of specification *S2*; multiple=*true* means that each instance of *S1Compo* must be wired with all the instances implementing *S2* and satisfying the constraints. When an instance of *S1Compo* will have to resolve that dependency, first Apam selects all the *S2* ***implementations*** satisfying the constraint *(apam-composite=true)*, and then Apam selects, all the ***instances*** of these implementations satisfying the constraint *(Scope=global).*
The dependency called *S2Single* is toward an interface. When it has to be resolved, Apam looks for an implementation that implements that interface, and preferably one instance satisfying (*x >= 10)*, any other one otherwise. A single instance of that implementation will be selected and wired.

Suppose that an instance *A-0* of implementation *A* is inside an instance *S1Compo-0* of composite *S1Compo*. Suppose that implementation *A* is defined as follows:

```
<implementation name="A" classname="…..A" specification="SX">
   <dependency interface="….I2" multiple="true" field="linux" id="toLinux">
      <constraints>
            <implementation filter="(OS=Linux)" />
      </constraints>
   </dependency>
   <dependency specification="S2" field="s2" id="fastS2">
      <preferences>
            <implementation filter="(speed &gt; 15)" />
      </preferences>
   </dependency>
```

Finally, suppose that specification *S2* provides interfaces *I1* and *I2* :

```
<specification name="S2" interfaces="….I1, ….I2"  >
      <definition name="OS" type="Windows, Linux, Android, IOS" />
      <definition name="speed" type="int" />
```

When instance *A_0* uses for the first time its variable *linux*, Apam checks if the *A_0* dependency *toLinux* is a dependency of its embedding composite. Indeed, *I2* is part of specification *S2*, and matches both dependencies *S2Many* and *S2Single* defined in *S1Compo*. However, *toLinux* being a multiple dependency, only *S2Many* can match the dependency, and therefore, Apam considers that *toLinux* has to be **promoted** as the *S2Many* dependency.

Because of this promotion, Apam has to resolve *S2Many* that will be associated with a set of *S2* instances matching the *s2Many* constraints (if any); then the same set of instances will be considered for the resolution of *toLinux*, therefore a sub-set (possibly empty) of *s2Many* instances will be solution of the *toLinux* dependency.
The *fastS2* dependency, being a simple dependency will be resolved either as as the *S2Single* instance, or as one of the targets of S2Many.

If, for any reason, an internal dependency is a promotion that cannot be satisfied by the composite, the dependency fails i.e. Apam will not try to resolve the dependency inside the composite.

A composite can explicitly, and statically, associate an internal dependency with an external one. For example, composite S1Compo can indicate

```
<promote implementation="A" dependency="fastS2" to="S2Single" />
<promote implementation="A" dependency="toLinux" to="S2Multi" />
```

It means that the dependency fastS2 of A is promoted as the dependency S2Single of S1Compo; in which case the constraints of fastS2 are added to the list of the S2Single dependency. It is possible to build, that way, static architectures as found in component models; however this is discouraged since it requires a static knowledge of the implementations that will be part of a composite, prohibiting opportunism and dynamic substitution.

Conflict access management: ConflictMan

By default, a service is used by the clients that have established a wire to it. There is no limit for this usage duration. Therefore, exclusive services (and devices) once bound cannot be used by any other client; there is a need to control service users depending on different conditions.

The wires are removed only when deleted (either setting the variable to null, or calling the release method in the API). When an exclusive wire is released, an arbitrarily selected waiting client is resumed.

Exclusive service management (from core).

An instance is said to be exclusive if it is in limited supply (usually a single instance), and cannot be shared. It means that the associated service can only be offered to a limited amount of clients, and therefore there is a risk of conflict to the access to that service.
In most scenarios, exclusive services are associated with devices that have the property not to be shared, as are most actioners.

```
<specification name="Door" interface=……
   singleton="false" instantiable="false" shared="false">
   <definition name="location" type="exit, entrance, garage, bedroom,…"/>
```

In this example, a device specified by “Door” is in exclusive access, but is in multiple instances (`singleton="false"` : a house may have many doors). It defines a property “location” i.e. the location of a particular door. `instantiable="false"` means that it is not possible to create instances of the Door specification, doors “appears”, i.e. they are detected

by sensors; and `shared="false"` means that a single client can use a given door (i.e. to lock or unlock it) at any given point in time.

Composite state management
The composite designer knows more about the context in which the components execute, than components developers, and can decide under which conditions a component can use a given exclusive service.

Apam distinguishes a property "state" associated to any composite. The state attribute is intended for managing exclusivity conflicts, its type must be an enumeration:

```
<composite name="Security" …
      …
   <contentMngt>
      <state implementation="HouseState" property=" houseState "/>
```

And implementation `HouseState` must define the attribute `houseState`:

```
<implementation name="HouseState" ….singleton="true" >
   <definition name="houseState" field="state" internal="true"
      type="empty, night, vacation, emergency, threat " value="night"/>
```

Each time an instance of composite `Security` is created, an instance of `HouseState` is also created and associated with the composite. That instance will be in charge of computing the composite state.
While this is not required, it is strongly advised to define the state attribute as an internal field attribute, in order to be sure its value will not be changed by mistake or by malevolent programs.

The own primitive
The own primitive in intended to enforce the ownership of instances. This is a critical importance since, in Apam, only the owner can define visibility and conflict access rules.

The own primitive enforces the fact that all the instances matching the declaration will pertain to the current composite. **The composite must be a singleton**.

```
<composite name="security" … singleton="true"
  <contentMngt>
    <own specification="Door" property="location" value="entrance, exit">
```

In this example, **all** Doors instances matching the constraint (`||(location=entrance)(location=exit)` appearing dynamically in the system, will be owned (and located inside) the unique `security` composite instance. No other composite instance can own Doors these Doors instances (and create them if Door would be instantiable).
In a composite declaration, a single own clause is allowed for a given specification (and all its implementations), or for a given implementation (and all its instance).
In the whole system, all the own clauses referring to the same component must indicate the same property and different values. This is checked when deploying a new composite. In case one of the own clause of the new composite is inconsistent with those of the already installed composites, (different property or same value) the new composite is rejected.

The Grant primitive.

The grant primitive is intended to enforce the resolution of a given dependency on some specific situations. In most cases, this dependency leads to an exclusive service (a device for example).
A grant primitive can be set only on dependencies with the wait behavior. It means that if the client is waiting for the resource, it is resumed as soon as the composite changes its state to the one mentioned in the definition and that it will not lose its dependency as long as the composite is in that state. However, when the composite leaves the state, the client may lose its dependency and can be turned in the waiting state.

```
<composite name="security" … singleton="true"
      …
<contentMngt>
  <own specification="Door" property=”location” value=”entrance, exit”>
    <grant when="emergency" implementation =”Fire” dependency=”door” />
    <grant when=”threat” specification=”break” dependency=”entranceDoor” />
   </own>
   <own specification="Door" property=”location” value=”garage”>
    <grant when="emergency" implementation =”Fire” dependency=”door” />
   </own>
```

In this example, when the (unique) instance of composite *security* is changed to enter the *emergency* state, the dependency called *door* of component *Fire* has priority on the access to the door target (an entrance or exit one only). To have priority means that if

- Component *Fire* (implementation or specification) tries to resolve the *door* dependency while *security* is in the *emergency* state, Apam gives to an instance of *Fire* the unique access to the door matching the constraint *(||(location=entrance)(location=exit))*. If not in the emergency mode, *door* is resolved as usually, and if no doors are available, the *door* dependency is turned into the wait mode.
- If the *door* dependency of component *Fire* is in the wait mode, when *security* enters the *emergency* state, Apam resolves dependency *door* towards its target (all the entrance and exit doors), even if currently used by another client, and resumes the waiting threads.

The system checks, at compile time, that all the grant clauses are defined against a different and valid composite state. Conversely, it is not always possible to verify, at compile time, that all the own clauses toward the same resource are defined on different values of the same property. This control is performed when a new composite is deployed or when a new composite instance is created; if another composite instance has a conflicting own clause, the new composite instance is rejected. Own clauses conflict if they are against the same resource, but on a different property, or on the same property but the same value.
However, for a completely deterministic behavior, it is advised to set granted implementation as singleton; otherwise, an arbitrary instance of that implementation will get the granted resource.

When *security* state changes to become *emergency*, Apam checks which doors owned by *security* (which includes those explicitly own, and may be others) are matching the *door* dependency. If these instances are currently wired by other client instances, these, their wires

are removed [4], and a *Fire* instance is wired toward the selected doors. When *security* composite leaves the *emergency* state, if instances are waiting for doors, one of them is selected, wired to the door and resumed.
In our example, if the house has an entrance or an exit door (that can be dynamically discovered), we know that the *security* will own them, and the *Fire* application is sure that it will be able to manages these doors in case of emergency.

However, the resolution fails, as usually, if the dependency constraints are not satisfied i.e. security does not own any door instance, or the owned doors do not satisfy the dependency constraints. If that case the grant primitive fails, and the system does nothing.

The start primitive
It is possible to create an instance of a given implementation, inside the current composite, on the occurrence of an event: the apparition of an instance (either explicitly created of dynamically appearing in the system).
This primitive has the same information as the instance primitive, but the event that triggers the instance creating in one case in the deployment of the bundle containing the instance declaration (for the instance primitive), while it is the apparition of an instance in the case of the start primitive.

```
<start implementation="S3Impl" name="s3Impl-int">
   <property name="S3Impl-Attr" value="val"/> <!-- Init attr value-->
   <dependency specification="S4"> <!—additional dependency constraints -->
      ….
   <trigger> <!—definition of the condition on which to start S3Impl -->
      <specification name="ASpec"> <!—an instance of ASpec appears -->
         <constraints>
            <constraint filter="(constraint on the instance)"/>
         </constraints>
      </specification>
   </trigger>
</start>
```

In this example, a new instance of specification S3Impl will be created when an instance of ASpec appears in the system (either created explicitly or dynamically appearing) . This primitive will be executed at most once (the first time an instance of ASpec appears after the S1Compo deployment).

[4] Warning: Apam removes the wire from the “old” client toward the exclusive instance, but if a client thread is currently executing in the exclusive instance, it will continue its execution. Therefore, the implementation of exclusive services should be careful not to retain the threads for “too long”. Exclusive services are supposed to perform “short” requests.
Note: a more satisfactory implementation would require the presence of proxies before the exclusive service, waiting the thread to leave the instance before changing the wires. It can be done later.