APAM: a service-based platform for dynamic and resilient applications

APAM

1. Introduction

In this document we present APAM, a service-oriented framework for the design, development and implementation of resilient and dynamic applications. APAM provides the following services:

1. Simplified design and implementation of service-oriented applications (over OSGi).
    - Simplification of the implementation of service-oriented applications, according to the "POJO" approach.
    - Simplified management of distributed applications.
    - Automated management of the dynamism and heterogeneity of the device.

In APAM, we consider that end users should not interfere in the administration or configuration of a system, or hardly ever. However, end users must be able to easily select and install on different devices (sensors, actuators, communication devices). APAM should detect and integrate devices in the existing applications, as much as possible in a transparent fashion; users should only contribute providing little information (the location of devices, for instance). End users can chose and install applications (the same way it would do with its mobile device), with the guarantee that they will be executed correctly.

APAM goal is to provide the resilience aspect to service-based applications, meaning that one application should continue to work despite of perturbations of any kind.

2. Resilience with respect to the context
    - Automatic integration of devices within the applications.
    - Control and dynamic architecture adaptation in response to the evolution of the execution context.

Most of the current service-oriented applications (and platforms that support them) make the assumption that any application can access all existing services and devices. Although, we consider that the platform supports the execution of a number of applications designed and developed independently, potentially by different suppliers, that must cooperate and share services and at the same time protect themselves. In particular, in home automation, applications share the knowledge of the "world" (the house and its contents, through its various sensors) and share the actions that can be performed in the "world" (through actuators, devices and screens), but the fact that actions can be performed by various independent applications can be a source of conflict.

3. Resilience with respect to conflicting applications, global coherence.
    - Composite definition, visibility and reuse rules.
    - Management of sharing and isolation among services and applications.
    - Management of access conflicts.
    - Consistency check and compatibility control among applications.

2. Simplifying the development of service-based applications

From a technical point of view, APAM is a service platform that extends OSGi, iPOJO and ROSE. APAM services run on one or more OSGi platforms; devices and services that are not OSGi (web service, hardware devices, legacy, ...) are represented by ROSE as OSGi services.

APAM extends the iPOJO approach, which aims to separate non-functional aspects from the source code (POJO = Plain Old Java Object), by injecting code (the "i" in "iPOJO") during compilation, which allows to automate the services mentioned above. Schematically APAM is a machine that manages dependencies among services: any access to a variable that refers to a service is captured by the injected code, which calls APAM to resolve this dependency in the current context and in conformance with the given policies and strategies. Similarly, any change in the environment can lead to change APAM dependencies already resolved, and modifying, if needed, the architecture of the running applications.

a. The APAM component model

The OSGi (and most service-oriented platforms) recognizes the concepts of service, package and bundle. A "service" is a Java instance that implements an interface (in Java context), a "package" is a Java package, and a "bundle" is a deployment artifact. There is no really a concept of component (the bundle concept take their place).

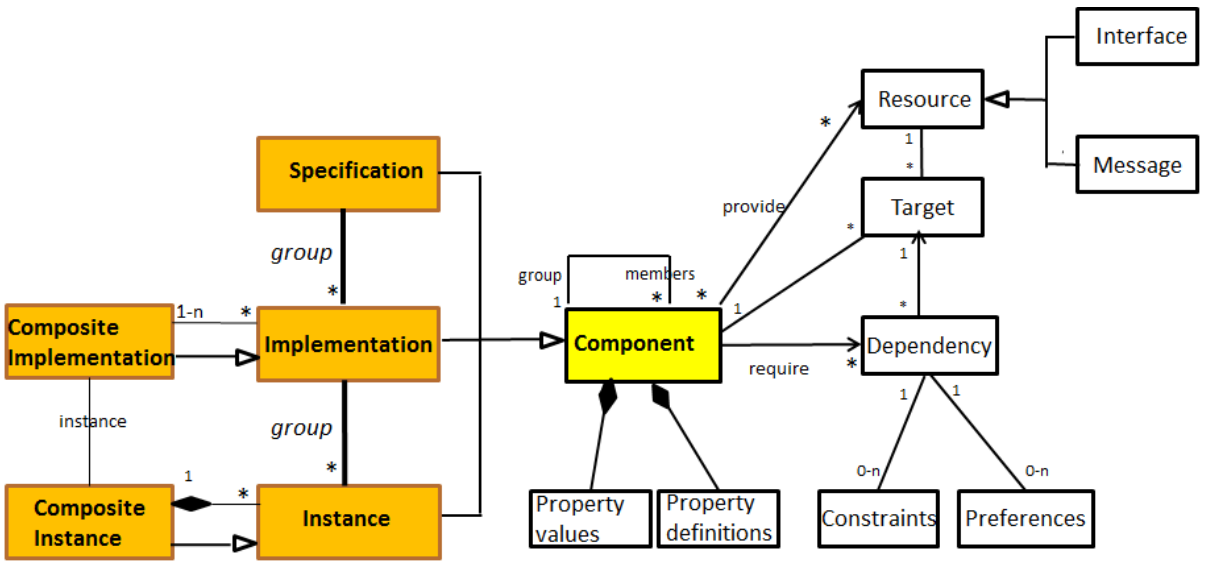
However, other models, such as SCA (Service Component Architecture) and other traditional component models, offer a strong-structured component concept: components (composites) can be made of other components (atomic or composite); composites are often hierarchical black-boxes with properties related to the level of abstraction, complexity control, protection, inheritance, etc. However, composites are usually defined statically during the development phase, or during the packaging phase at the latest, which makes complex their dynamic modification and adaptation.

On the other hand, approaches built on top of OSGi, such as iPOJO and SpringDM, offer simple component models; but they do not specify how to compose components (composition remains dynamic, according to the service vision of OSGi) and do not provide a structuration level. These models are able to simplify the development on top of OSGi (which is their main goal), but not to structure applications or ensure their consistency at runtime.

The APAM component model aims to provide:

- the flexibility of service-based platforms (composition and dynamic adaptation)
- guarantees of application consistency and better runtime control (strong type),
- structuration tools that allows to create hierarchical black-boxes, white-boxes and gray-boxes (composites).

The APAM metamodel is the following:



APAM considers three types of primitive components: specifications, implementations and instances that are specializations of the abstract type "component". All component:

- has properties,
- declares properties,
- provides resources (interfaces and messages), and
- declares dependencies to resources or other components.

Properties are tuples <attribute, value> where *attribute* is the property identifier, and *value* a singleton or a set conforming to the declared type. Property declarations are tuples <attribute, type, default-value> where type is a primitive type (integer, string, boolean, enumeration), or a set type whose elements are strings or enumerations. See ***Property management***.

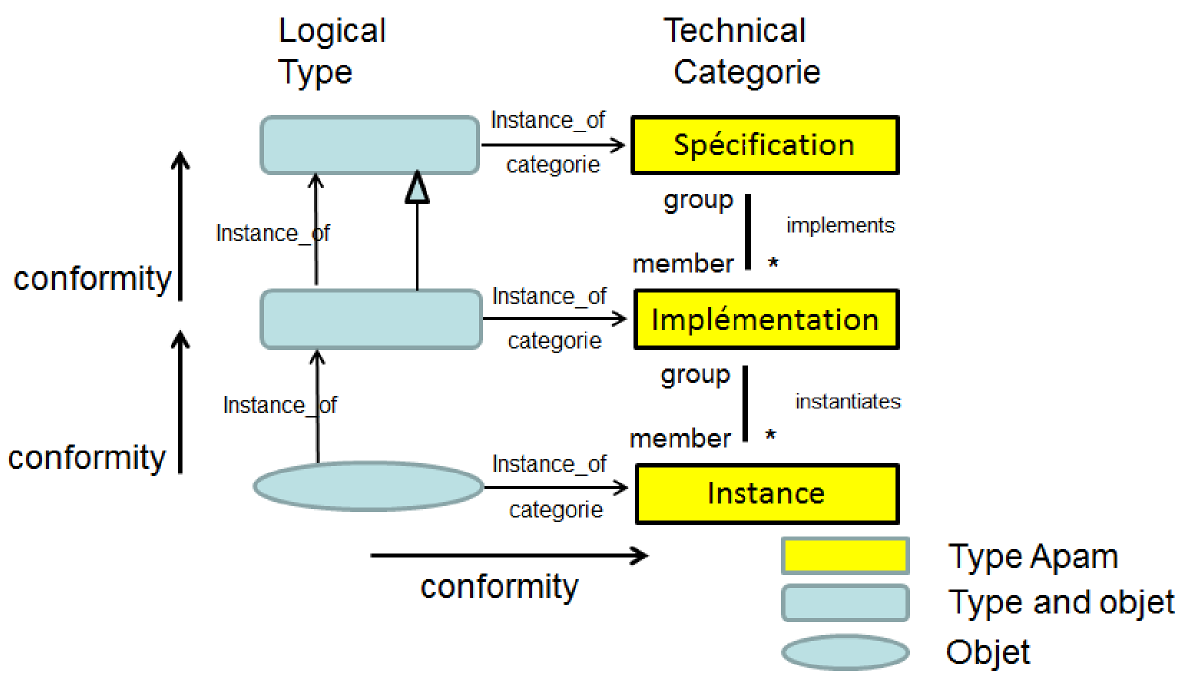
A component can provide resources that are either interfaces or messages.

A dependency definition is a declaration, from a component, of the components or resources that it might require during its execution. At runtime, dependencies are the effective relationships between components, usually calculated dynamically, see ***Dependency resolution and extensibility*** and ***Dependency management and resolution strategies***.

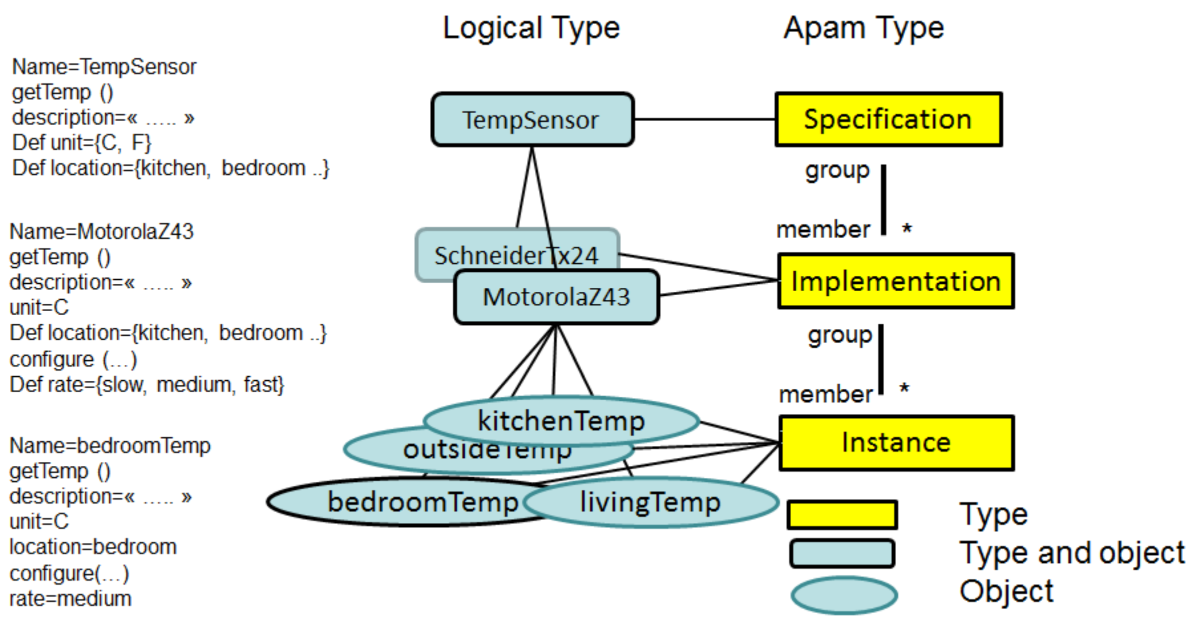
b. Groups

Strong-types are based on the concept of equivalence group. An equivalence group is a tuple <group, members> where *group* is a component and *members* a set of components which have all the characteristics defined by the *group* component, instantiate the properties declared by the *group* component, and can have additional properties. The relationship between a *group* and its *members* is similar to the relationship between a class (group) and its instances (members). Common properties are the class variables (static in Java), and the group properties are the instance variables.

In APAM, types consider a double nature of any component: a technological nature (specification, implementation and instance) related to Java and to the underlying service platform; and a business nature, related to a specific application domain. The technological nature is imposed by APAM and their service platforms; the business nature is open to developers.



Technological and business conformances (conformance by instantiation) enable strong-types and a recursive group mechanism that allows flexibility and multiple concretization levels. The group mechanism is a generalization of the materialization and powertype concepts. An example is the following one:



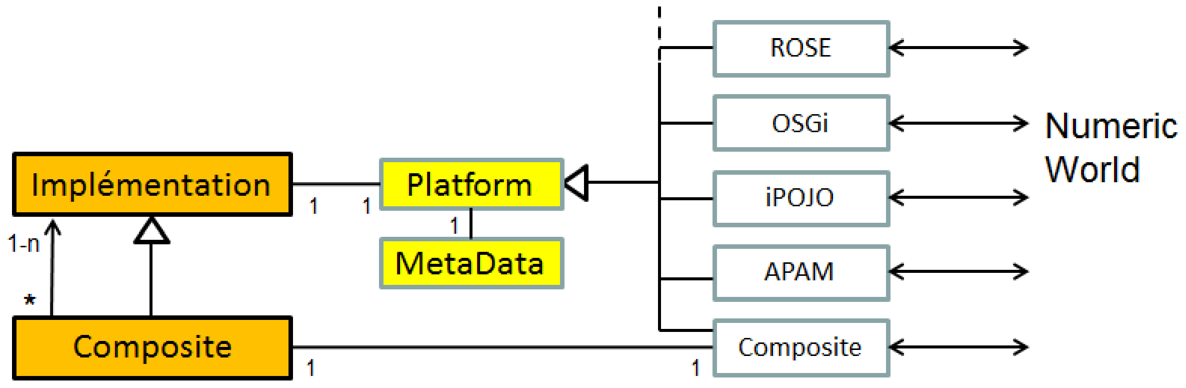
In this example, *capteurTemp* is both an *APAM specification* and the definition of a specific type (temperature sensor). By the instantiation relationship with the *APAM specification*, *capteurTemp* is a specification in the technological sense (e .g. a Java interface in a package in a bundle); and from the business perspective, it is the definition of a business concept (a *getTemp* function, a property *description*, the definition of the properties *unit* and *location*).

MotorolaZ43 is both a member of the group *capteurTemp*, and an *APAM implementation*: it is then an implementation of a temperature sensor. It has all the properties expected from an implementation (a Java class into a bundle, dependencies, …), and all those expected from a temperature sensor (it implements the method *getTemp*, has the *description* inherited from *capteurTemp*, …). *MotorolaZ43* is also a subtype of *capteurTemp*, it is then a type that can declare new values and new properties: in this example, the method *configure* and the property *rate*.

Bedroom is a member of the group *MotorolaZ43* and an *APAM instance*. It has all the properties of the APAM instances (a Java object of the *MotorolaZ43* class, and an OSGi service), and it is a materialization of *MotorolaZ43*, it has all the properties (*description = …; unit = c*) and instantiates the definitions (*location = bedroom, rate = medium*). An *APAM instance* is not a group: instances are not types, they are simple objects.

This mechanism of declaration and instantiation of components at multiple abstraction levels allows performing static type-checking without having to know the objects that will exist at runtime. For example, the following filter: "capteurTemp ((unit = c) and (location = kitchen))" is checked at compilation time and it is correct; the filter "capteurTemp ((unit = celsius) and (piece = kitchen))" produce two errors at compilation time: *celsius* is not a valid value for the property *unit*, and *piece* is not declared as property.

APAM provides several specializations of the technological concepts. Thus, an implementation can be associated with a native APAM implementation, iPOJO, Rose or OSGi (other types of implementations can be offered later), and it can be atomic or composite[1].



Thus, *MotoralaZ43* can be a legacy code OSGi, an APAM component, a specific driver with a specific protocol (e.g. Zigbee), a remote service, etc. without interfering with its description and its business properties. The associated class (iPOJO for example) is responsible for computing the meta-data out of the information available in the associated technology, and to synchronize the value and behavior of the Apam object, with the value and behavior of the associated object(s) in the associated platform.

c. Dependency resolution and extensibility

APAM is a machine extensible through managers. Three classes of managers are defined: dependency, dynamism and property managers. The APAM standard distribution provides several managers, including various dependency managers. Thus, and contrary to iPOJO or Spring which

[1] Powertype and concretization concepts do not allow specializing the “basis class”, in APAM this is a fundamental requirement to handle and extend technological concretizations. Consequently, we have defined the group concept.

resolve service dependencies using the services (instances) running on the current OSGi machine, APAM can resolve a dependency in various ways. Using the default provided managers, a service dependency can be resolved:

- by selecting an existing APAM service (imposed APAMMan),
- by instantiating an APAM implementation (imposed APAMMan),
- by selecting an existing legacy service (extension OSGiMan),
- by deploying services from local and remote repositories (extension OBRMan)
- by resolving on remote APAM machines (extension Distriman)
- by "preemption" of services already used (extension ConflictMan)
- or any other strategy according to the extensions defined by developers.

The declaration of an implementation indicates the "instrumented" fields of the class, i.e., the fields that will be managed by APAM. An instrumented field is a dependency declaration. At runtime, when accessing a not initialized instrumented field, APAM tries to resolve the dependency, i.e. to find (or create, or deploy, ...) an instance that satisfies the dependency declaration. The disappearance of a service sets the corresponding field to the "not initialized" value, a new resolution will be attempted at the next use of this field, which can select a new provider, initialize a new implementation, deploy a new service, use a remote service, etc.

See ***Dependency management and resolution strategies,*** *page* ***23***

d. Managing dynamism: the dynamic managers

The resolution mechanism allows APAM reacting to changes on the state of a system (appearance, disappearance, property changes of sensors and services) by modifying transparently and dynamically its architecture in order to ensure, as far as possible, its normal functioning. This is the resilience property.

A dynamic manager specifies the expected behavior when resolutions fail, and when services appear or disappear. The default dynamic manager, Dynaman, allows setting in “wait” a service for which a dependency could not be resolved, or throw an exception. On the arrival of a service, Dynaman unlocks the services waiting for that service, re-launch the corresponding applications, etc.. Other business behaviors can be defined through specialized dynamic managers.

See ***Dependency management and resolution strategies,*** *page* ***23***

e. Sensors, actioners and other devices

Any device using a discovery and communication protocol managed by Rose, appears as a legacy OSGi service. It is therefore considered by APAM as a normal service and managed as such.

See Rose documentation.

f. Distribution and distributed applications

APAM provides two ways for defining and executing distributed applications. The first one consists in using only Rose, allowing thus to describe explicitly and statically the services exported and imported by Rose, and to use the Rose proxy generation mechanism to communicate explicitly with remote services.

The second solution consists in using a dependency manager that manages distribution. This is the case of Distriman manager. With Distriman, the distribution becomes transparent to applications; a resolution can then look for a service on the local machine (APAMMan, OSGiMan), deploy the service from a known bundles repository (ObrMan), look for the service on a remote machine (Distriman), or even deploy the service on a remote machine from a known repository (Distriman).

The machines that will be visible at runtime are not known in advance, and neither their known repositories. Machines can appear and disappear dynamically; Distriman relies on a discovery mechanism to detect the machines that appear and disappear. In case of disappearance, for example, a using remote service becomes unavailable. In the next access to that service, APAM executes the regular resolution mechanism. Distriman allows therefore to easily writing dynamic and resilient distributed applications and resilient, which is a main goal of APAM.

See ***Distriman: a distribution manager***.

3. Application architecture

a. Encapsulation: the composite concept

Most of the programming languages and component models define the concept of composite with strong encapsulation properties. Thus, a component contained into a composite is not visible outside the composite, and conversely, it cannot see the components outside the composite. These composite are often referred to as black-box composites.

These characteristics have important properties because the encapsulation is an abstraction: the composite "hides" and "protects" its components. The behavior of a composite is independent from its execution context, improving reuse, allowing isolated testing and ensuring that the composite behavior is the expected one in all circumstances.

The composite appears as an atomic component, allowing nesting several abstraction levels and defining architectures at each level. The visible complexity is related to the number of elements of the considered abstraction level. Black-box composites allow managing the scaling factor and controlling complexity.

These properties are at the origin of modern computing; it is then surprising that service-based platforms like OSGi do not offer facilities for the structuration and encapsulation of services[2]. Indeed, in OSGi, any service can potentially see all the other services, and can be used by any service. The Java protection mechanisms allow denying the access to some services (packages, classes, resources, etc.), the OSGi protection mechanism allows masking services, but they do not offer the structuration or visibility concepts. In addition, services being shareable by default, a same object can be used by multiple threads of different applications; it becomes then extremely difficult to know "who" works this object and even less the applications for which a called service works[3]. Finally, the opportunistic service resolution ensures that a service will be connected to a provider arbitrarily chosen from the set of available providers. Controlling an application running on a OSGi platform that contains other

[2] Modularity and control of imports/exports in OSGi apply to packages and bundles, not to services.
[3] This is why the Java protection systems must inspect the call stack, which is very expensive.

applications is in general difficult. In theory, using Java security enables such a control, but at a high complexity and execution costs non-negligible.

b. Dynamism

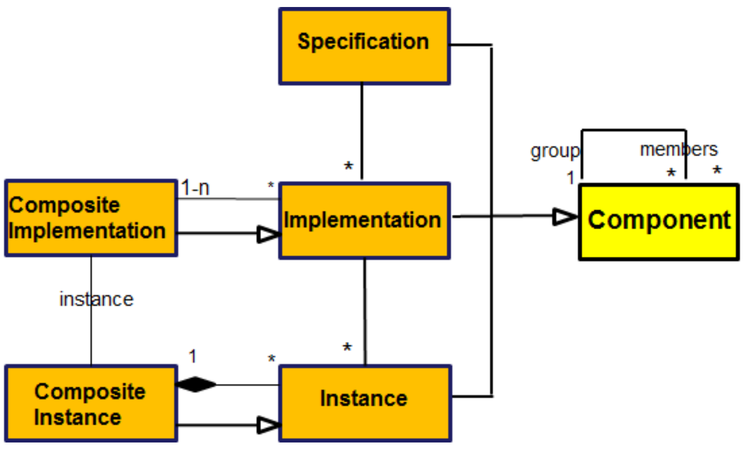
From the point of view of dynamism, the situation is inverted. Component models require, in general, giving the list of components of a composite and defining statically all the connections. It becomes impossible, or difficult, to change dynamically the composition of a composite. This is a serious limitation in dynamic environments. Thus, the “traditional” composites are not suitable.

For their part, service-based platforms have been designed for dynamism and sharing.

Basically, composites express a concept of ownership of well-known components: it is a closed and static world; while services express the opportunistic sharing of functionalities offered by anonymous providers; it is an open and dynamic world.

c. APAM composites

APAM composites try to consolidate these two seemingly irreconcilable visions. The goal is thus to enable the creation of classical black-box composites, the opportunistic use of the services shared by anonymous providers or any intermediate option.



APAM distinguishes composite implementations, which are implementations containing other implementations, and composite instances, which are instances containing other instances. An executable composite implementation must define its provided resources and an atomic implementation (referred as the main implementation) which provides at least the same resources provided by the composite implementation. A composite instance is an instance of a composite implementation, where its main instance is an instance of the main implementation of the composite implementation.

Declaring Components, *page* ***18,*** shows how to declare components.

By default, composites do not indicate their content (except for their main implementation): the content of composites is dynamically built by the resolution mechanism. Thus, when a client instance s belonging to a composite instance cs from CS (a composite implementation) asks for a service resolution:

1. If an existing instance p is found: s is connected to p, whatever the composite instance containing p (if p is visible, see the next section).
2. If an implementation P is found (and visible) but it does not have available instances, an instance p of P is created within cs.
3. If an implementation P is found in a repository, P is deployed and placed within the composite implementation CS, and an instance p of P is created within cs.

Note that the resolution mechanism does not differentiate between an implementation P atomic or composite, or between an instance p atomic or composite.

We distinguish between logical and physical deployment. Physical deployment implies loading a bundle in an OSGi platform. Logical deployment indicates that the wanted implementation exists but it belongs to a composite CF that it is not visible (APAMMan fails at step 2), OBRMan is called and it finds a bundle containing P (the step 3 is successful) but the bundle is already deployed; APAM act as if P comes to be deployed by CS; P belongs then to both composites CF and CS. An implementation can then belong to multiple composites (those who have deployed it, physically or logically).

Conversely, an instance belongs only to the composite that creates it; instances being atomic or composite, composite instances can be nested; the structure of composite instances is like a tree. Thus, APAM manages a forest where each tree is an application. An application may not declare a composite, a root composite to contain the application will be created on the fly; this allows the execution of legacy applications without modifications.

Components (atomic or not) being used in various composites and applications, are designed to define only intrinsic properties, i.e., properties related to the source code and to the utilization hypothesis expressed by the component developers, independently from the execution context. The intrinsic properties are true regardless the use made of that component, and intrinsic constraints must be verified regardless the use made of that component.

Composites are designed to define contextual properties for its contained components, i.e., properties that will be only true and constraints that will be only verified when the component belongs to this composite. It is then possible to define contextual dependencies (see ***Contextual dependencies***, *page* ***29***), contextual constraints (see ***Contextual constraints,*** *page* ***30***) and visibility rules.

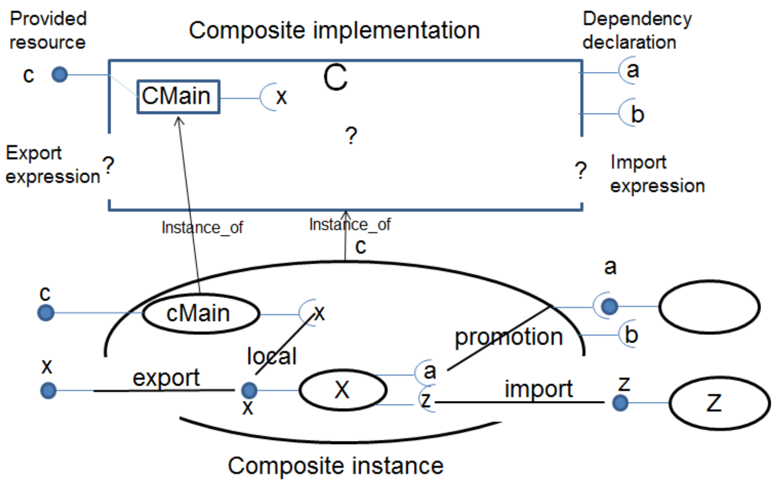
4. Managing concurrent applications

a. Visibility: from encapsulation to sharing

A main role of composites is to define the wanted isolation and sharing levels. A composite can be a black-box if it does not export its components and if it does not import other components. There is then a complete encapsulation, such as in the classical component models, without having to list statically all the contained components. They can be dynamically deployed and dynamically instantiated. This allows, among other things, having third-party applications completely self-contained and isolated from the rest of the system. These applications must be deployed in advance or have a repository containing the different components that will be dynamically selected regarding the needs and the current execution context.

Conversely, a composite can be a white-box if it exports and imports everything (this is the default strategy of service-based platforms). This type of composite, referred to as opportunist, uses when possible the available services (exported/provided by others), and deploys services that are not yet available making them available to other composites.

APAM provides a flexible way to define, for each composite, the imported and exported components. See ***Visibility control,*** *page* ***31***.



In the example of the figure above, a composite implementation C declares its provided resources (here the interface c), the required resources (here the interfaces a and b), and the main implementation (here CMain). C can give a logical expression (a LDAP filter) in order to define the imported and exported components (all components are exported and imported by default).

At runtime, the dependency x of cMain was resolved by creating an instance X which provides the interface x. This is a local resolution, because cMain and X are in the same composite, and a local component is always visible. By cons, if X satisfies the export expression, x is visible from the outside of composite C. Dependencies a and z of X must be resolved. Because a is an explicit dependency of C, the dependency a of X is promoted in the dependency A of C, resolved in the context of C. In this example, we assumed that there exists an instance Z providing z and which verifies the import condition; X is then connected to Z which remains outside c.

If expressions are always false, the result is hierarchical black-boxes; if expressions are always true, the result is a flat system where all services are visible. According to expressions, all the intermediary options are possible.

b. Visibility vs security

The platform must support the concurrent execution of various independent applications that cooperate and share services, and ensure the protection of the source code of applications and the safety of their data.

The visibility rules presented in the previous section are a structuration mechanism that allows both application modularization and management of service sharing. The visibility mechanism structures the content of the service registry, allowing accessing to services in a finer way than a flat register like the OSGi registry. However, the visibility rules do not constitute a protection mechanism: any visible service can be potentially used by any client.

To define the access control policies, APAM relies on the standard Java protection mechanisms. Concretely, when resolving a dependency (see section ***Dependency resolution and extensibility***), APAM checks that the client code has the needed permissions to approach the required service; APAM follows the OSGi security specification and uses ServicePermission for validation.

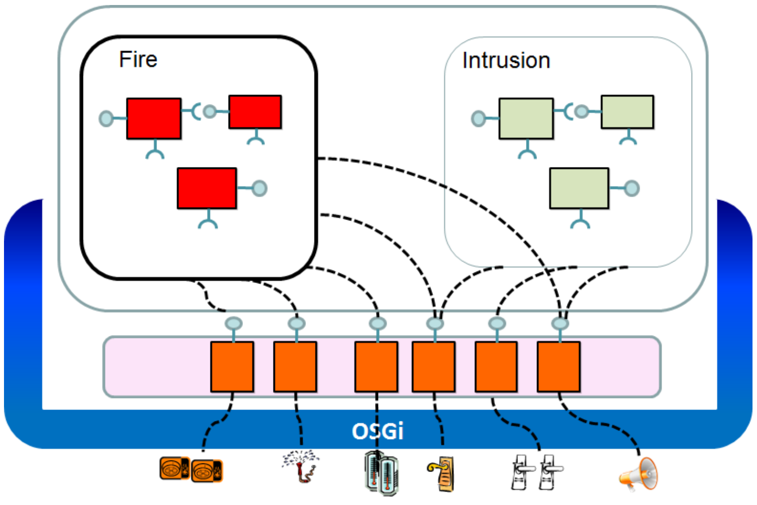
Visibility and access control in APAM use different specific mechanisms, but complementary and orthogonal. These two mechanisms are not necessarily addressed to the same actors. The visibility rules are specified by the application providers within its declaration; the access policies are specified by the manager of the platform (or gateway) via the own platform mechanisms (see security deliverable).

c. Control of conflicts of concurrent access

The control of sensors and actuators demands additional considerations in terms of sharing and conflict management. Typically, these devices are deployed independently of the applications running on the platform, and are meant to be shared and used by several applications. Nevertheless, their non-controlled concurrent use may produce the malfunction of applications and pose risk to users.

APAM aims to provide a device sharing control transparent for the application developers. Devices are reified and accessed as normal services; the conflict management is defined outside the application in a declarative way.

To illustrate the problematic and the APAM proposed mechanisms, consider the simplified example (from [4]) of a fire protection application and an intrusion detection application which control the actuators in a home automation environment. These two applications have been developed by different vendors and ignore each other.



For the developer of each application, the devices are accessed transparently as services, using proxies that encapsulate the specific network protocol. The fire protection application uses the temperature and smoke sensors to detect a fire, and controls the sprinkler heads and the opening of house's doors. The intrusion detection application uses the presence and motion sensors to detect intruders, and controls the doors in order to lock the house access.

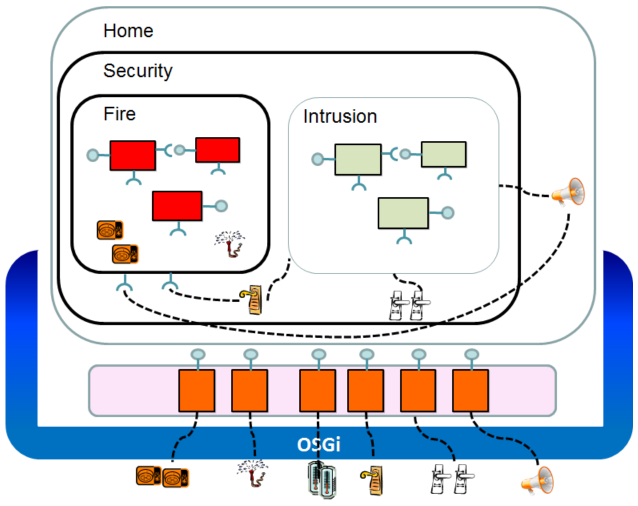
In this example, we can observe that some devices:

- are private to a particular application (e.g., the sprinkler heads);
- are shared and do not conflict (e.g., the sound alarms);
- are shared and potentially conflict (e.g., the door locks).

Notice that the application developer cannot anticipate these scenarios, because it is not aware of the other applications that will be deployed on the gateway. Each application provider must thus develop its application without making assumptions about possible conflicts, as if he/she had the exclusive control of the devices.

We have defined the concept of "silo" that is a collection of applications that share the same functional domain and potentially the available devices. In the house, we could find silos such as security, energy, comfort, media, etc. Silos are materialized by composites whose mission is to define the policies of protection, visibility, sharing and management of critical devices. A silo must own the devices and services of which it must ensure a consistent use. The choice of silos and their goals is a global decision (related to the house ontology).

For the example of home security applications, we define the silo "Security", shown in the figure below, which defines that the smoke detectors and the sprinkler heads are private to the fire protection application (or silo "Fire"), and that the doors and sound alarms are shared by silos "Intrusion" and "Fire".



To do so, we must ensure that the silo "Fire" owns the sprinkler heads and the smoke detectors, and that the silo "Security" owns the doors. This is defined by the "owns" primitive (see section ***The own primitive***, *page* ***35***).

```
<Composite name=”Fire" >
    <contentMngt>
              <owns specification =”SmokeDetector”/>        <!—Fire owns all the smoke detectors -->
              <owns specification =”Sprinkler”/>            <!—and all the sprinkler heads -->
             <export  instance=”false”/>                     <!—and it does not share them -->

    </contentMngt>
    <dependency interface ="Alarm”>                          <!—Fire requires an alarm -->
    <dependency interface ="Door" id=”doors” >               <!—and doors -->
       <constraints>
          <instance filter=”(location=entrance)”>            < !—but it only needs the entrance door -->
        <constraints/>
    </dependency >
</composite>
```

Any instance, and then any physical device, can belong only to a single composite; it is task of such a composite to defining the policies for conflict management. A device (or an instance) is considered as private or shared depending on the visibility rules of its composite. Although devices can be physically accessible directly on the network, with the “owns” clause it is possible to impose a structuration that allows restricting their access and usage.

Inside the composite, devices (and instances) are visible for all the applications. It is possible to define that a service can only have one user at a time (shared = “false”) and that a device must be assigned exclusively to an application according to specific situations. In our scenario, we specify that in the presence of fire the fire protection application is priority and must have the exclusive control of doors; if an intrusion is detected, the intrusion detection application is priority; and in all the other cases the doors can be controlled by any application.

In order to express this policy, APAM introduce two concepts: composite state (see section ***Composite state management***, page ***35***) and exclusive service allocation (see ***The Grant primitive*** page ***36***).

The state of a composite synthetizes the current execution context of the applications contained in the composite. It is calculated by a specialized component which observes the execution of applications and determines the global situation. Once the composite state determined, it is possible to specify, for each device controlled by the composite, who is the priority client for the current state. In the example, we can specify the following policies for controlling the house’s doors.

```
<Composite name=”Security">
  <contentMngt>
     <state implementation="HouseState" property=" houseState "/> <!—definition of the state and the component that
handles it -->
    <own specification="Door" property=”location” value=”entrance, exit”> <!—rules for the entrance and exit doors -->
      <grant when="emergency" implementation =”Fire” dependency=”door” />  <!—Fire uses the door on emergency -->
      <grant when=”threat” specification=”break” dependency=”entranceDoor” /><!—Break uses the door on intrusion -->
    </own>
</contentMngt>
</composite>
<implementation name="HouseState" ….singleton=”true” >
  <definition name="houseState" field="state" internal="true"
    type=”empty, night, vacation, emergency, threat ” value=”night”/>     <!— possible state values -->
<implementation>
```

These rules specify the access order to devices over time, depending on the state of its owner composite. When a client application will use a device, APAM is asked to resolve the dependency to the respective service. If there is a priority client in the current state of the composite that contains the device, the application is staged (see ***Dependency management and resolution strategies***). When the composite changes of state, if there is a client waiting that matches the “grant” primitive for the new state, APAM will preempt the service access to the current client and will unlock the priority client.

Notice that the various APAM mechanisms for managing conflicts are orthogonal and complementary. When a client application accesses a device, APAM successively checks that all the following conditions are satisfied:

- the device is visible to the client (i.e. the device is local or importable)
- the device is local or exported by its owner composite,
- the client owns the required access permissions,
- in the current state of the owner composite, the device is allocated to the client.

d. Consistency control and application compatibility

The definition of conflict management policies requires a global knowledge of the involved silos, the device types and the application types that can be hosted in the main silo; “Security” in our example. This may seem contradictory to the vision of an open environment in which the user can freely install new devices and new applications; but in fact, the existence of these policies allows both flexibility and insurance of a consistent operation.

Indeed, conflict management policies can be declared in terms of component specifications, and not of their concrete implementations. For example, the policy that we have shown for the security domain remains valid even if we do not know which fire protection application will be effectively deployed by the platform user, nor the concrete devices that will be installed or discovered in a particular house. Other fire protection applications and other devices can be deployed later, even on the fly, without requiring changing any definition.

The ability to define abstract policies for access management allows analyzing and reasoning statically about the possible access conflicts in a particular application domain without having to know the concrete implementations of applications and devices. From the design phase, it is possible to validate and check the access policies, from the definition of the component specifications. The consistency of the execution related to the specified policies is ensured in APAM by the conformance relationship between the different component abstraction levels (specification, implementation and instance) via the group mechanism (see ***Groups***, page ***6***).

Notice that it is not necessary to know exhaustively in advance all the applications contained in a silo. In the presented example, there is possible of deploying new applications into the “Security” silo, which can for example control the doors in non-critical states. It is therefore possible to add new applications into an existing silo.

The declarative definition of a policy for conflict management captures a generic and partial knowledge of an application domain. This allows defining a flexible and consistent configuration space, allowing users to install dynamically new applications and devices.

Part II. Annex

1. Compilation

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.

2. Declaring 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.

a. Specifications

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.

b. Implementations

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.

c. Instances

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).

d. 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.

3. 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.

a. 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[4].

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>
```

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

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.

b. 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.

c. 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.

4. 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()`).

5. 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.

6. 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 (an instance) finds the service provider (an instance) it requires. A resolution is launched when the client uses a variable of the provider type; if the resolution is successful, the client java variable is loaded with the address of the provider.

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.

a. 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.

b. 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.

7. 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.

8. Constraints and preferences

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 in order to make precise their requirements:

```
<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" />
```

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.

9. 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.

10. 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.

11. 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) $\wedge$ import(cc, p)
2. visible (c, P) $\wedge$ import(cc, P) $\wedge$ 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.

a. 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.

b. 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[5]. 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) $\wedge$ (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.

12. 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>
```

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

```
      <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.

13. 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.

a. Exclusive service management

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.

b. 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.

c. 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.

d. 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 [6], 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.

e. 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).

[6] 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.

14. Distriman

Distriman is a dependency manager which tries to resolve a dependency looking at the other APAM machines which are currently visible. During a resolution, Distriman can ask the remote visible APAM machines to resolve the dependency. If the remote APAM succeeds, Distriman creates a proxy in the local machine connected to an end-point on the distant machine, and return the created proxy as the solution of the resolution. Note that the remote resolution can involve OBRMan and therefore a remote deployment, but not a remote Distriman to avoid hubs.

Therefore, transparently, a service can be connected to another service on a remote machine, and/or can involve a remote deployment. Distriman listen to the arrival and departure of APAM machines and reacts to a departure by removing the local proxy, which will start a new resolution ending, maybe, in selecting a service running on another machine …

a. Principles

The characteristics of Distriman are the following.

- Distriman reifies all the visible APAM machines as a composite which name and properties are those of the distant machine. This composite represents the distant machine and contains the remote implementations that have been imported.

- Distriman interprets a model which expresses, for each composite, which are the dependencies that can be resolved remotely and which are the components that can be exported towards other APAM machines.

- Importing a service is similar as deploying that service. Importing a service creates :

    - An APAM implementation with the same name and properties as the original implementation, but « instantiable=false ». This implementation being « deployed » is contained in the client composite type, and in the composite which represents the distant machine. Distant implementations and their clones are immutable: it is not possible to change their properties or definitions.

    - An APAM instance, on the client side, with the same name and properties as the original instance. The original instance can already exist, or can be created, depending on the remote composite resolution process. The local instance being created pertains to the client composite instance, and therefore has the visibility defined by that composite instance. The instance can be modified, in local as well as in distant. If properties of either instance are modified, both instances are deleted and recreated to enforce their value synchronization.

    - A proxy (locally) and an end-point (remote). The proxy is the local instance serviceObject.

b. Distribution Model

A Distriman model contains the definition of the import and export of one or more composites.

```
<distriman>
   <composite type-name=”Expr”>
      <import specification="true" machExp=”Exp” install=”Exp” />
```

```
        <import implementation="A*-lib" machExp="Exp" install="Exp" />
        <export specification="A*-lib" | implementation="Exp"/>
    </composite>
    <composite type-name= …..
</distriman>
```

`Expr` is an LDAP expression, or « true » or « false ».

The model is associated with a bundle and describes the Distriman strategies for the composites contained in that bundle.

If, for a given composite, no export is provided, that composite is not visible from outside the current machine. If, for a given composite, no import is provided, that composite dependencies will be resolved only inside the current machine.

<import This tag expresses that a distant resolution is required if

- The source of the resolution pertains to the current composite, and
- **specification= "Exp" | implementation= "Exp" | interface= "Exp" | message= "Exp"** The resolution target is matching the content (i.e. the target is respectively of the type specification, implementation, interface of message) and its name matches the expression.
- **machExp = « exp »** expresses that the target resolution must be intended on the remote machines that satisfies the expression. The expression is evaluated locally against the properties available on that machine representative. If the selected machine does not owns a component satisfying the dependency constraints, another machine is selected, until a satisfactory component is selected or all machines are tried. If no solution is found, returns null. If more than one machine matches the expression, they are tried in a random order. If *machExp* is missing, *machExp="true"* is assumed (i.e. all visible Ampam machines).
- **Install = « Exp »**. If no resolution is found, *install* expresses the condition under which a remote deployment can (and must) be intended. The expression is evaluated against the properties of the machine representative. If the expression is satisfied, the resolution is intended, OBRMAN enabled, on the corresponding distant machine.
  If *install* is missing, *install=false* is assumed (no remote deployment).

<export specification= "Exp" | implementation= "Exp"

This tag indicates which components of the current composite are visible from (exported to) other APAM machines. Any implementation that matches one or the other expressions is visible. By default all implementations are visible. Only the instances with a global visibility are exported.

References

Our publications

[1] H. Cervantes, R. Hall. "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model". In Proceedings of the International Conference on Software Engineering, 2004-05-01, ICSE Edinburgh, Scotland.

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

[3] D. Moreno-Garcia, J. Estublier. "Model-driven Design, Development, Execution and Management of Service-based Applications". SCC, Hawaii USA July 2012.

[4] J. Estublier, G. Vega. "Managing Multiple Applications in a Service Platform". Proceeding PESOS: In. Workshop on Principles of Engineering Service-Oriented Systems, at ICSE Zurich, June 2012.

[5] Jacky Estublier, German Vega and Elmehdi Damou. "Resource Management for Pervasive Systems". Proceeding WESOA International Workshop on Engineering Service-Oriented Applications. At ICSOC, Shanghai, 12 October 2012.

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

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

[8] J. Estublier, Idrissa Dieng, Eric Simon, Diana Moreno. "Opportunistic Computing. Experience with the SAM platform". Pesos, Cape Town, at ICSE 2010.

Specifications techniques

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

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

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

[12] OSOA (2007).Service Component Architecture: Assembly Model Specification Version 1.0.:

Conflicts management

[13] H. Jacob, C. Consel, N. Loriant. "Architecture Conflict Handling of Pervasive Computing Resources". IFIP Int. Federation of Information Processing 2011. LNCS 6723, pp92-105.

[14] S.K.S. Gupta, T. Mukherjee, K. Venkatasubramanian. "Criticality Aware Access Control Model for Pervasive Applications". ICPCC 2006.

[15] V. Tuttlies, G. Schiele, C. Becker: "Comity - conflict avoidance in pervasive computing environments". In: International Workshop on Pervasive Systems (2007)

[16] R. Sandhu, E.J. Coyne, H.L. Feinstein, C.E. Youman. "Role Based Access Control". IEEE Computer. 1996, pp38-47.6

[17] D. Massaguer, M. Diallo, S. Mehrotra, and N. Venkatasubramanian, "Middleware for pervasive spaces: Balancing privacy and utility," in 10th International Middleware: ACM/IFIP/USENIX, ser. LNCS, 2009, vol. 5896, pp. 247–267.

[18]

Visibility and protection

[19] D. Retkowitz, , S. Kulle. "Dependency management in smart homes". In: Senivongse, T., Oliveira, R. (eds.) DAIS 2009. LNCS, vol. 5523, pp. 143–156. Springer, (2009)

[20] L. Fiege, M. Mezini, G. Mühl, A. P. Buchmann. "Engineering Event-based Systems with Scopes". Proceedings of the European Conference on Object-Oriented Programming (ECOOP'02), LNCS 2374, Malaga, Spain, Springer-Verlag, June 2002

[21] 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.

[22] Pedro Capelastegui1, Olga Gadyatskaya, Fabio Massacci, and Anton Philippov. Security-by-Contract for the OSGi platform. Technical Report # DISI-12-002. http://www.disi.unitn.it

Matérialisation et Powertypes

[23] E. Zimanyi, A. Pirotte, and T. Yakusheva, “Materialization : a powerful and ubiquitous pattern abstraction,” pp. 630–641, 1994.

[24] [80]M. Dahchour, A. Pirotte, and E. Zima, “Materialization and Its Metaclass Implementation,” vol. 14, no. 5, pp. 1078–1094, 2002.

[25] [81]C. Atkinson and T. Kühne, “The essence of multilevel metamodeling,” «UML» 2001—The Unified Modeling Language. …, 2001.

[26] [82]J. J. Odell, Advanced Object-Oriented Analysis and Design Using UML. Cambridge University Press, 1998.

[27] [83]C. Gonzalez-Perez and B. Henderson-Sellers, “A powertype-based metamodelling framework,” *Software & Systems Modeling*, vol. 5, no. 1, pp. 72–90, Nov. 2005.

Component models and architecture

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

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

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

[31] 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

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

[33] 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.

[34] 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

[35] 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.

[36] 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.

[37] 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.