Resource Management for Pervasive Systems

Jacky Estublier, German Vega, and Elmehdi Damou

Grenoble University - LIG, 220 rue de la Chimie 38041 Grenoble BP53 Cedex 9 France
{Jacky.Estublier,German.Vega,Elmehdi.Damou}@imag.fr

Abstract. In pervasive contexts, many different applications, from different providers, will compete for access to resources: physical resources like sensors and actioners, as well as for software resources (services). Sensors provide information about the state of the world, and actioners change the world which can put goods and persons at risk. At least for safety reasons, it is critical to closely control, at any point in time, and in all circumstances, which service(s) are using which resource(s).

Pervasive systems face the difficult challenge of providing some safety, reliability and resilience properties, verified at design and compile time, while executing in many different configurations unknown statically, with dynamic services and devices, competing for resources with unknown applications and facing unpredictable configuration changes. This challenge can be seen from two perspectives: how to design and develop pervasive applications in such a demanding context; how to execute these applications while satisfying the requirements despite the unpredictable context and changes.

This paper discusses the requirements for future pervasive gateways and presents the Apam dynamic service middleware. Apam interprets at run-time a formalism describing the desirable behavior of a system, and enforces this behavior in a very wide range of unplanned configurations while resisting the many changes that may occur.

Keywords: Service Oriented Computing, Service Selection, Service Composition, Composite services, Software engineering environments.

1 Introduction

The wide diffusion of cheap and wireless devices makes it possible for many spaces, public or private, to be populated by communicating devices. Typically, in the house, it is envisioned that the set-top box will play the role of the “universal” residential gateway, supporting many downloaded applications from, for example, the android market place. These applications ignore each other but still compete for the access to the available resources. With respect to “usual” software applications, pervasive applications have at least the three following unusual characteristics:

1. All the applications share the same physical world,
2. Applications must be installed, with zero configuration, in many different contexts,
3. Applications must tolerate unplanned changes that occur during execution.

A. Ghose et al. (Eds.): ICSOC 2012, LNCS 7759, pp. 368–379, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Although these characteristics, individually, can be found in other domains, their concomitance in pervasive systems constitutes a very serious challenge, so far unresolved. Solving each one of these points requires addressing a number of challenges:

Sharing the Same Physical World. This is probably the most demanding and far reaching issue. For example, in the house, the common world is the house itself, with its equipment and inhabitants. The current state of the world is perceived through many sensors, and it is changing by itself or through actioners. These devices are shared by the different applications running in the house; there is a need to manage the access to these devices, and to avoid conflicts. When applications ignore each other, the global behavior can become inconsistent and unpredictable, which is of critical importance since it is the real world that is changed, and therefore it can put goods and persons at risk. Clearly, an application cannot solve all these issues, because applications are usually designed by different groups of developers that ignore each other, and therefore they can hardly synchronize themselves. This issue requires a high level dynamic middleware that has the capability to control the whole system, made of many applications competing for the same resources and to enforce a "consistent" and "safe" execution of independent applications which can interfere in almost unpredictable ways.

Zero Configuration. These applications are supposed to be bought, installed and run by end users. The market of domestic applications is expected to grow fast, such that the number and variety of applications will rapidly be very large. The challenge, here, is that the whole system must adapt itself, without human intervention, to the different configurations that can be found in the different houses. Since the applications will be developed independently, the challenge falls on the shoulders of the system designers that will have to describe the overall desirable system behavior, without a complete knowledge of the execution context (device, services, and applications).

Unpredictable Changes. Many pervasive applications will run for very long periods of time (e.g. heating control) during which almost any change can occur. These changes are "normal" when it concerns the state of the house (e.g. mobility and actions of their inhabitants) but also, the end user can install or uninstall applications from very large application market place. These changes are unpredictable, both in time and nature. The challenge is that the running system and its running applications must dynamically adapt to these changes. The new devices and applications must be integrated into the current systems without compromising the stability, continuity and consistency of the whole system.

Applications being designed independently, the challenge falls again on the designer's shoulder: he/she has to express which changes are allowed during execution and how they can be integrated such that the new system still satisfies the overall desirable behavior. Addressing these challenges require addressing two different dimensions:

- **The design** point of view, with formalisms capable of describing the "desirable behavior" of the whole system without the complete knowledge of the actual execution context, supporting a large range of unplanned and undefined devices and applications, and adapting to many unplanned dynamic changes.
- **The execution** point of view, interpreting and enforcing the design formalism: enforcing in all circumstances the overall desirable behavior.

Our approach is architecture based. We distinguish the **design architecture** which is an abstract description of the characteristics that all actual architectures should satisfy; and the **execution architecture** which is the actual state of the world, in terms of devices and applications. The design architecture is in terms of abstract services (specifications) and containers (composites) which define scopes and visibility rules. The execution architecture is in terms of service instances (devices are reified as service instances too) and "wires". We have defined a conformity relationship between design and execution architectures such that a large and potentially infinite number of execution architectures can be conforming to the same design, including the dynamic changes.

The paper is illustrated by a scenario in section 3. Section 4 shortly describes the Apam components, section 5 illustrate the dynamic management, and section 6 the protection mechanism: the composites. Section 7 presents the conflict handling strategies; section 8 summarizes and section 9 concludes the paper.

2 State of the Art

The pervasive domain is both in the self-adaptive (autonomic, context aware) and resource management domains. The first one considers a single application in a fluctuant context [1][3], while the second considers multiple applications conflicting on stable resources. We have multiple applications running in a fluctuant context and conflicting on variable resources.

From the design point of view, resource conflict detection and management is an issue in many domains, and has been addressed in many different ways. A static analysis approach, using dedicated languages or model checking, is very powerful, but makes the hypothesis that the applications and at least the devices are statically known. In pervasive computing, each house is potentially different; devices and applications are dynamic, therefore static analysis is not sufficient.

Maybe the oldest way to manage conflicts is using Access Control Lists (ACL) and Role Based Access Control (RBAC) [8] recently adapted to the pervasive context [6] adding a "criticality" status, but as a conceptual model only.

Most propositions in pervasive computing identify a special global attribute (called state [4], mode [5], criticality [6], context [7] …) and describe the conflict resolution with respect to the value of this attribute. In [7], each application statically defines its actions on the real world and what it considers to be a conflict. Then in each physical space, a conflict manager compares the descriptions and computes if conflicts can happen. This approach moves most of the burden onto the application developer's shoulders, and does not solve the issue of different and incompatible visions of what a conflict is. In [9], it is the user privacy which is the central issue.

It is possible to consider conflict handling as a special case of dependency management, taking also into account the current "state", and priorities. In [10], AOP and "exclusive binding" are used, while [4] propose a DSL in which exclusive actions, ACL, priorities and required resources are defined; from this DSL, conflicts are detected and code is generated. However these approaches are preliminary and ignore

many device management issues; dynamic evolution is not really supported since any change requires recompiling, regenerating and redeploying the whole system. A protection based on scope and visibility control, close to our work, can be found in [11], but limited to event based systems and not addressing resources access control.

Our approach can be qualified as architecture based dependency management. In contrast to most approaches, we introduce an architecture in which composite entities encapsulate their content in order to provide a scope for 1) defining the dynamic and conflict management policies and 2) to control the visibility of services. Every level of the architecture has its own "state" and its own policies defined in terms of abstract services that will be mapped to concrete service instances at runtime, allowing a large range of unexpected evolutions, both in terms of devices and new applications.

3 A Scenario

For illustration purpose, suppose that a home gateway supports a number of applications including a security manager which manages fire and intrusion threats. Intrusion itself is based on both movement detection and breaking and entering. The house is supposed to be equipped with many devices, including various smoke detectors, sprinklers, motion and break-in detectors, alarm, and doors that can be locked or unlocked. Alarm and doors require exclusive access. In case of fire, the entrance door must be unlocked (by the Fire application), but at night, the entrance door should be locked (by the Intrusion application). The alarm should be used by the application that needs it. Furthermore, the house owner can install new devices and download new services at any time.

At design time, it would be nice to produce a specification of the system, at the highest possible level of abstraction. Such a specification should contain the design architecture, including the aspects necessary and sufficient to describe the "desirable behavior" of the system. The concepts used in the design architecture should be those established in "house ontology". In our case this design could look as follows, in which rectangles stand for the specification of sub-systems, devices or services.

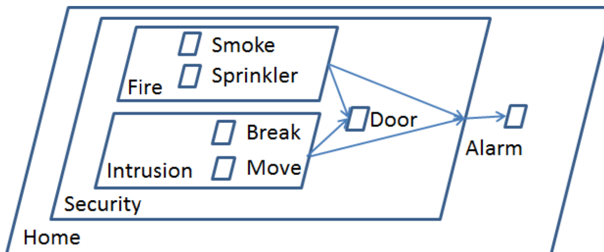

Fig. 1. A possible design architecture

While this figure summarizes roughly our scenario, it is essentially semantic free; the semantics of the specifications is undefined, and it does not give any information about how our challenges are addressed which are to make sure that the different applications (or sub-systems) will perform as expected, despite the fact (1) they ig-

nore each other, (2) they may conflict on some devices (what about a fire at night?) and (3) new devices and applications can appear at any time, with new conflicts.

In substance, our goal is to define a formalism which gives a similar high level vision of the system, but semantically rich enough to include the characteristics expressing how the challenges above are addressed, and allowing to detect, at design and compile time, the inconsistencies and potential conflicts. To that end, the concepts of specification have been made formal, including the resolution process that, at run time, transforms a design architecture into an execution architecture such that the running system behaves in conformity with the design architecture. Let us first introduce the Apam components.

4 Primitive Components: Context Free Behavior

Apam Components are defined at three levels of abstraction: specification, implementation and instances. Specifications mostly define the resources provided and required by the component, and the Java classes containing the provided interfaces. An implementation is a piece of code (Java classes in our system) that implements one specification, (i.e. it provides and requires the resources defined by the specification). An instance is a running Java object in the platform.

Apam relies on the POJO (Plain Old Java Object) approach in which the source code of a component should only be concerned with the application logic. Each component is associated a metadata (in XML currently); at build time that metadata is interpreted and the POJO is transformed into an Apam component (an OSGi bundle). Let us first show how the Fire manager, at the specification and implementation levels, can be described in our system.

```
<specification name=”Door" interfaces=”home.…Door” exclusive=”true”/>
<specification name=”Alarm" interfaces=”home.…Alarm exclusive=”true”/>
<specification name=”Sprinkler” ….

<specification name=”Fire" interfaces=”fr.imag….FireStatus”>
   <dependency specification="Alarm"/>
   <dependency specification="Door" id=”doors”>
   <definition name=”hasSprinkler” type=”Boolean” default=”false” />
</specification>

<implementation name=”FireSprinkler" specification=”Fire”
classname=”fr.imag….FireMng”>
   <dependency field=”alarm” id=”Alarm”/>
   <dependency field=”door” id=”doors” />
   <property hasSprinker=”true” />
   <-- Additional dependencies -->
   <dependency specification=”Sprinkler” field=”sprinklers”/>
   <dependency specification=”Smoke” field=”smokeDetectors” />
</implementation>
```

The example above shows the device specifications (`Door` and `Alarm`, ..), including the interfaces by which they can be managed and with the property `exclusive` meaning that such a device can have at most one client. Specification `Fire` declares the interface it provides `FireStatus`, and its dependencies towards specifications `Alarm` and `Door`. Dependencies have a unique id, by default it is the name of the associated speci-

fication; in the example, the dependency toward `Door` is called `doors`, and toward `Alarm` it is called `Alarm`. The line `<definition … hasSprinkler` is the definition of a property that the implementations of `Fire` can instantiate. Specifications are really components; at design and build time they are compiled and packaged as OSGi bundles containing the interfaces and the metadata; they are packaged, stored and deployed exactly as implementations.

The implementation `FireSprinkler` indicates that it implements specification `Fire`, and therefore provides and requires the same resources. It must indicate which class (`FireMng`) implements the interface `FireStatus`, and which (Java) fields in this class are the dependencies `Alarm` and `doors` defined in the specification; the type and cardinality of fields `alarm` and `door` are found in the source code. `FireSprinkler` has additional dependencies toward specifications `Sprinkler` and `Smoke` because this specific fire manager uses smoke detectors and sprinklers to perform its job. To make this clear, that implementation sets the attribute `hasSprinkler`, defined in the specification to `true`. Other implementations could use other ways, (like only using the alarm when the temperature is too high) this is why these dependencies are not in the specification. This metadata information is stored in files inside the eclipse project in which is developed the associated Java code; it is interpreted transparently during Maven[1] build by a specific Maven plug-in that injects byte code for dependency management and builds the corresponding OSGi bundle.

5 Composites: Context Dependent Behavior

A primitive Apam component (its source code and metadata) does not make any hypothesis about its context of use, the availability and dynamic behavior of resources, or any hypothesis about possible conflicts. Therefore primitive components are easier to program and as reusable as possible. However, Apam requires additional (meta) information to manage the consistency of the system seen as members of an ecosystem (both i.e. actors and subjects of that ecosystem). In Apam, this information is included in the Design Architecture of the system by means of the concept of Composite Component[2]. A Composite component is an actor in the ecosystem defined as a number of connected components (a sub-system, an application). A composite captures the shared knowledge about the ecosystem (expected devices, dynamism) required to express the expected global system behavior in that context; in particular its relationships with the other composites, its protection and conflict management policies.

Usual service platforms have a flat structure, which is very inconvenient because any service can use any other one as soon as it knows its published interface; for example, any service could lock any door at any time: it is scary! A protection mechanism is needed. Apam is based on the concept of composite component, as a

[1] `http://maven.apache.org/`

[2] This is a simplified description of the Apam composite concept ; for more detail, see [12].

mechanism for protection applying the concepts of dynamic architecture, scoping and visibility to pervasive systems.

The main protection mechanism is based on visibility control. Suppose that a client instance *x*, pertaining to composite instance *cx*, asks for a provider of specification *Y*. The client instance *x* pertaining to *cx* can see *y* pertaining to composite *cy* if

- *y* pertains to *cx (cx* = *cy*) or
- *cy* lends *y* to its friends, and *cx* is a friend of *cy*, or
- *cy* lends *y* to the application, and *cx* and *cy* pertain to the same application,
- *cy* lends *y* to the whole platform.

cx is a friend of *cy* if a *friend* relationship is established from *CY* [3]to *CX*. An instance pertaining to a single composite instance, the instances in a platform are organized as a forest. An application is defined as a tree (i.e., a root composite instance); *cx* and *cy* pertain to the same application if they pertain to the same instance tree.

A composite can define which instances can be lent to other composites using the tags *local*, *friend* and *application* The value of these tags is an expression to be applied to instance properties. An instance cannot be lent if it matches the *local Instance* expression; it can be lent to friend composite instances if it matches the *friend Instance* expression; it can be lent to any composite of the same application if it matches the *application Instance* expression; and finally it is lent to the whole platform if it matches none. If it matches more than one expression, the most restrictive one is assumed.

Symmetrically, a composite designer must be able to decide whether or not to borrow the instances lent by other composites. For this purpose, can be specified the tag *borrow Instance*=<*expression*>. If the requested resource matches the expression, the platform must try to borrow an instance if it exists. If the expression is not matched, an instance must be created. By default, the expression is false, i.e., by default composite should use their own instances. Let us illustrate on our scenario, in which `Fire` is now a specification composite and `FireCompo` is an implementation composite.

```
<Specification name="Fire" interfaces="fr.imag….FireStatus">
   <dependency specification="Alarm"/>
   <dependency specification="Door" id="doors">
   <definition name="hasSprinkler" type="Boolean" default="false" />
   <state type="{normal, onFire}" value="normal" >
</specification>
<composite name="FireCompo" specification="Fire" main="Fire" >
   <dependency specification="Door" id="doors" >
       <constraint filter="(location=entrance)" />
   </dependency >
   <contenMngt>
      <dependency specification="Fire" >
         <preference filter="(hasSprinkler=true)" />
      </dependency>
      <owns specification="{Smoke,Sprinkler}"/>
      <local instance="(exclusive=true)" />          <!-- private. -->
      <borrow instance="false" />                    <!—only use external dep-->
   </contenMngt>
</composite>
```

[3] Lower case like *cy* are instances and upper case like CY are implementations.

The Fire specification composite has a state with two values: `normal` and `onFire`. `FireCompo` first refines dependency `doors` as a door satisfying the constraint `location=entrance`. The `FireCompo` composite, must define its “main implementation”, which is any atomic implementation that provides at least the same resources as the composite. Here it must be an implementation of specification `Fire`, that preferably satisfies the constraint `hasSprinkler=true` (`FireSprinkler` is a possible resolution). The `<owns ..>` tag indicates that the services implementing specifications `Smoke` and `Sprinkler` must be owned by the `FireCompo` composite. Apam checks, at compile time and when an application is about to be deployed, that a single composite has a `owns` clause on a given service.

`<local instance="(exclusive=true)"` indicates that the exclusive services it owns, the sprinklers in our example, cannot be lent; they are only to be used by this composite; but the other owned services can be lent freely, for example the smoke detectors. `<borrow instance="false"` indicates that the components inside this composite can only use the components owned by this composite and those explicitly declared in the composite’s dependencies (i.e. the alarm and the entrance door). With these declarations, the overall architecture, at instance level is the following:

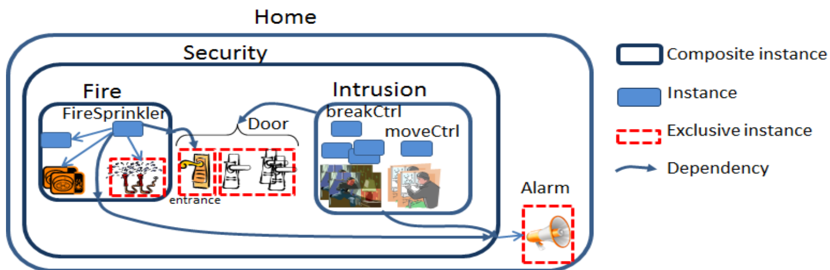

Fig. 2. The instance architecture

Instantiating a composite implementation consists in creating an instance of its main implementation. The main implementation instance will also call its dependencies; by the resolution process Apam will look for a “visible” instance satisfying the constraints. In our example, since the composite declared `borrow instance=false`, are visible only the instances owned by the composite, those that can be deployed from the composite repository, and those visible through the explicit dependencies. For example, `FireSprinkler` can only use the sprinklers own by `FireCompo` (it cannot borrow them). The only way to use a service located outside the composite is through the composite dependencies. When field `doors` is used for the first time, Apam realizes that it is an external dependency, and therefore tries to resolve the dependency from `FireCompo` toward `Door` which turns out to be an entrance door. Once this dependency resolved, Apam resolves the dependency from `FireSprinkler` to `Door` as a subset of `FireCompo` dependencies: `FireSprinkler` will only use the entrance door while it is programmed to manage any door. We say that the `doors` dependency of `FireMain` is promoted as the `FireCompo` doors dependency.

6 Conflict Handling

The ownership control and the fine grained dependency management solve many potential conflicts, but the most serious one remain. In our example, a door can be locked or unlocked, and it is declared `exclusive` which means that a single service can use it at a time. Simultaneously, the `Intrusion` composite declared it requires the door since it has to lock it when the house is empty, while the `Fire` composite declared that it needs to open it in case of fire. Not being explicitly owned, our mechanism would simply give the door to the first one that asks for it, and non-determinism would follow.

In our philosophy, the door control is the responsibility of the composite that owns it; in our case the `Security` composite. To that end, the `Security` composite defines its possible states: `Normal`, `Empty`, `Intrusion`, and `Emergency`.

```
<specification name="Security" interfaces="ccc" main="securitySpec">
   <state type="{Normal, Empty, Intrusion, Emergency}" />
   <owns Specification="Door" />       <!-- all doors.    -->
   <local instance="true"/>            <!-- lends nothing.-->
   <start component="Fire" specification="{Smoke, Sprinkler}" />;
   <start component="Intrusion" specification="Motion" />;
   <start component="Intrusion" specification="Break" />;
   <grant component="Fire" dependencies="{Alarm, doors}"     when="Emergency";
   <grant component="Intrusion" dependencies="{Alarm, Door}" when="Intrusion";
   <grant component="Intrusion" dependencies="Door"          when="Empty";
</specification>
```

The `Security` composite owns the doors; it is therefore entitled to decide which application can make use of them. In this example, the entrance door is granted to the `Fire` component when the state of the security component is `Emergency`, i.e. when a fire is detected. Granting the entrance door to `Fire` means that the door is pre-empted, i.e. if the door is currently used by another service, that service is turned into the "wait" mode, its connection is removed, and a connection is created between `Fire` and the door. The connection is resumed when the condition (`Emergency`) is no longer satisfied. The when clause only contains values of the current composite state, which allows to check, at compile time, that an exclusive service can only be granted to, at most, one component in each possible state. The capability to automate conflict detection at compile time when assembling large systems, and at execution time when a new application is about to be deployed, is an important property of the system. In this example the alarm and the entrance door are allocated (granted) to the `Fire` component in case of `emergency`; the door is allocated to the `Intrusion` component when the house is empty, and the doors are available to any service othewise. Only applications inside the security area can ask for a granted access.

7 Designing in a Unpredictable World: Adapting to Changes

A contribution of this work is the definition of a pervasive system at the specification level, i.e. defining a system with partial knowledge, through an architecture containing only the fundamental conditions for a "normal" behavior. Suppose, for example,

that the house owner downloads a new and unknown application making use of sprinklers. In our example, it is not the `Fire` specification that owns the sprinklers, but the composite implementation `FireCompo`. If the `Fire` application does not explicitly owns these devices, they can be owned by other application, and the `Fire` application may be denied their use when needed. To avoid this risk, the designer could decide to allocate the management of smoke detectors and sprinklers to `Fire` at the specification level and the break and movement detectors to `Intrusion`. We end up with the picture in Fig. 2 but with precise executable semantics.

With this design, the compiler can check, at specification level, if there is any risk of access conflict and any inconsistent declarations, without any need to indicate the implementations potentially used (and potentially unknown). This design is irrespective of the implementations that will be used, the devices that will be discovered, and the new applications that can be downloaded. For example, downloading, into the Home composite, media applications (sharing the alarms for example), health applications, or different implementations (and vendors) of intrusion or fire managers, adding any kind of device, the fundamental behavior of security, fire and intrusion will be enforced. In the same time, these new applications and devices will be integrated into the system, as long as the `grant` and `owns` primitives are not inconsistent. In our example, the smoke, break and movement detectors, as well as doors and alarms, can be used by these new and unknown applications, without compromising the consistency of the whole system.

8 Implementation

The APAM platform includes a complete design and execution environment. The design environment is an extension of Eclipse [**Error! Reference source not found.**]; components (atomic and composite) are developed and described in XML files. The Apam compiler, which is implemented as a Maven plugin, is transparently executed during the build phase; it checks the architecture validity and builds the bundles.

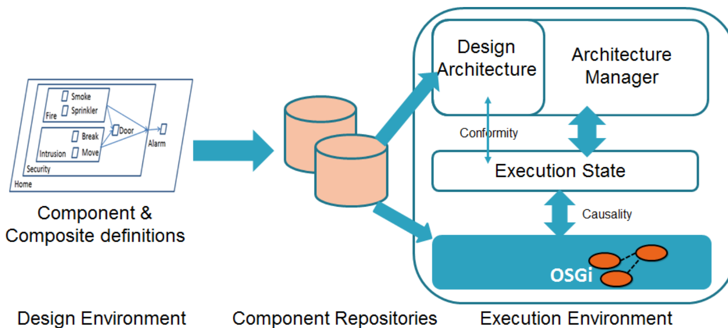

Fig. 3. APAM implementation architecture

Once the application design architecture validated, it is provisioned as bundles into component repositories and it is available for execution. The design architecture is a first class artifact managed at design, deployment and execution time.

The APAM runtime includes iPOJO[2] and a standard OSGi platform in which the services and devices drivers are making up the application. Apam builds an execution state representing the current execution architecture, including specifications and composites. The execution state is a causal model. The architecture layer manages the execution state such that its evolution will be conforming, at all times, to the design architecture [Fig3].

The execution environment is designed to be very efficient; component interactions are implemented as direct object invocations, and the APAM runtime is only triggered when an unresolved dependency is invoked for the first time. Modifications of the architecture are reflected by modifying the injected references of the components. With respect to OSGi, the memory overhead is about 10%, essentially due to the reified state, and the execution efficiency is similar to OSGi. In comparison, component frameworks like JEE or SCA are orders of magnitude slower than APAM.

9 Conclusion

Software engineering's best practices require that each individual application must be developed as if running alone with all the resources it needs; but on the other side, applications in pervasive environments must face an unknown context, unpredictable dynamic changes and must compete for its resources with unknown applications.

Our solution consists in structuring the complete system resources (applications, services and devices) inside composites. The composites being themselves components, the structure can be nested (through the ownership relationship), but also any directed graph (through the dependency and friend relationships). Each composite is in charge of declaring the strategies to be applied to the components it owns and its relationships with the unknown outer world. The strategies discussed in this paper include the dynamic behaviour of resources, the architecture management, and the access conflicts. Composites being components also come at the three levels of abstraction: specification, implementation and instance, corresponding roughly to the design, development and execution phases of a software system life cycle.

We believe that the key point is the capability to design a pervasive system at the specification level in which is described only the management strategies to be applied to the abstract resources owned and required by each service. At that level, there is no need to provide the exact nature, number and availability of the resources that will be used by the system at run-time; because it is not needed, not known or because it may change during execution. The implementation architecture, defined in the composite implementations, can refine the specification architecture.

The second key point is that these strategies are declared in each composite, and the composite architecture is known. The inconsistencies and risks of conflict can be detected, either statically at compile time when assembling large systems, or just before deployment in case of unplanned evolution: strategies are enforced in all cases.

The third key point is that the resolution process transforms at run-time the design architecture (in terms of specifications and dependencies) into execution architecture

in terms of instances and wires. The execution state changes only when a resource appears or disappears, or when a wire must be established or deleted. Apam computes the next state of the system such that it is a valid architecture i.e. for a new wire, the selected instance is found in the current state or an implementation is found in an available repository. In all cases, the new state satisfies the dynamic, access control and visibility strategies expressed in the design and implementation architectures.

The design architecture is abstract enough to fit a wide range of execution contexts, and a very large (possibly infinite) number of execution architectures. The lazy nature of Apam resolution makes that the transformation from design architecture to execution is performed "just in time" within the actual context and system state, enforcing both architecture conformity and adaptation to the current context.

We believe that we have met our challenge of checking statically and enforcing dynamically the resource management strategies without the complete knowledge of the available resources and their dynamic behavior; and adapting to the unpredictability of dynamic changes and competing applications.

References

1. Cervantes, H., Hall, R.: Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In: Proceedings of the International Conference on Software Engineering, May 1. ICSE Edinburgh, Scotland (2004)
2. Escoffier, C., Hall, R.S., Lalanda, P.: iPOJO: an Extensible Service-Oriented Component Framework. In: IEEE Int. Conference on Services Computing, USA (July 2007)
3. Zhang, W., Hansen, K.: Semantic Web based Self-management for a Pervasive Service Middleware. In: IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems (2008)
4. Jakob, H., Consel, C., Loriant, N.: Architecturing Conflict Handling of Pervasive Computing Resources. In: Felber, P., Rouvoy, R. (eds.) DAIS 2011. LNCS, vol. 6723, pp. 92–105. Springer, Heidelberg (2011)
5. Mukhija, A., Rosenblum, D.S., Foster, H., Uchitel, S.: Runtime Support for Dynamic and Adaptive Service Composition. In: Wirsing, M., Hölzl, M. (eds.) SENSORIA. LNCS, vol. 6582, pp. 585–603. Springer, Heidelberg (2011)
6. Gupta, S.K.S., Mukherjee, T., Venkatasubramanian, K.: Criticality Aware Access Control Model for Pervasive Applications. In: ICPCC 2006 (2006)
7. Tuttlies, V., Schiele, G., Becker, C.: Comity - conflict avoidance in pervasive computing environments. In: International Workshop on Pervasive Systems (2007)
8. Sandhu, R., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role Based Access Control. IEEE Computer, 38–47 (1996)
9. Massaguer, D., Hore, B., Diallo, M.H., Mehrotra, S., Venkatasubramanian, N.: Middleware for Pervasive Spaces: Balancing Privacy and Utility. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 247–267. Springer, Heidelberg (2009)
10. Retkowitz, D., Kulle, S.: Dependency Management in Smart Homes. In: Senivongse, T., Oliveira, R. (eds.) DAIS 2009. LNCS, vol. 5523, pp. 143–156. Springer, Heidelberg (2009)
11. Fiege, L., Mezini, M., Mühl, G., Buchmann, A.P.: Engineering Event-Based Systems with Scopes. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 309–333. Springer, Heidelberg (2002)
12. Moreno-Garcia, D., Estublier, J.: « Model-driven Design, Development, Execution and Management of Service-based Applications. In: SCC, Hawaii, USA (July 2012)
13. Estublier, J., Vega, G.: Managing Multiple Applications in a Service Platform. In: Proceeding PESOS: Workshop on Principles of Engineering Service-Oriented Systems, at ICSE Zurich (June 2012)