A Graph-based Approach for Contextual Service Loading in Pervasive Environments

Amira Ben Hamida and Frédéric Le Mouël and Stéphane Frénot and Mohamed Ben Ahmed
{amira.ben-hamida, frederic.le-mouel, stephane.frenot}@insa-lyon.fr,
{mohamed.benahmed}@riadi.rnu.tn

INRIA Amazones, CITI Lab., INSA Lyon
21 Avenue Jean Capelle
F-69621 Villeurbanne Cedex, France

Abstract. The pervasive computing paradigm promises great abilities whenever and wherever a user goes. However, as people are shifting from the desktop to more resource-constrained devices, issues due to scarce resources may appear preventing from the use of the available services and applications.
In this paper, we consider the adaptive deployment as a mainstream solution to suit service-oriented applications to different context constraints such as the users requirements, the hosts resources, the services properties and the surrounding environments.
We put forward a graph-based deployment approach for service-based applications so as to make these applications adaptable to the runtime contextual constraints. We introduce the AxSeL architecture, A conteXtual Service Loader in which services and their dependencies are represented as a bidimensional graph. The dependency graph is then coloured through a process taking into account the devices, services and users constraints. This process aims to choose to load or not a service according to its execution context.
A prototype based on Java and OSGi technologies is implemented in order to demonstrate and evaluate our approach.

1 Introduction

For several years, the applications deployment on local devices was considered as a monolithic process in which the application is first installed and executed, then periodically updated through an access to the provider site.

Since the inception of pervasive environments, users have been moving from desktop machines to mobile handheld constrained devices. This evolution introduced an important need of flexibility and adaptation of the used applications to the context that is no longer static. Applications need to adapt to user mobility, resource changes, user preferences, the surrounding environments, etc. Hence, the need of dynamic, adaptable and context-aware applications.

Adapted design models as service oriented applications [1] are based on paradigms as services registration, description, discovery, reuse and separation

of concerns, which allows great abilities of functionality spread in distributed environments and dynamic contextual adaptation.

In this paper, we address applications deployment and execution on mobile devices in pervasive environments. We adopt a hybrid approach of local and remote deployment of service-oriented applications which allows their use even if the network is disconnected. Besides, we believe that using the service-oriented paradigm allows great abilities of loading the applications on different devices, of dynamically enriching the applications with new functionalities, of switching on the fly between functionalities and of reconfiguring the loaded ones.

We introduce the AxSeL architecture, a conteXtual Service Loader that enables the service-oriented applications autonomic loading and adaptation on resource-constrained devices. In AxSeL, an application is represented as a bidimensional dependency graph of services and components. We take advantage from the graph theory in order to easily deal with services runtime dependencies and to benefit from existing approaches of optimal graph walks. Moreover, in order to achieve context-awareness listeners are used to capture and interpret incoming context events to generate the optimal deployment actions. According to context events (service property change, service appear/disappear, device resource change), AxSeL generates 'snapshots' (views) of the current runtime application. In this paper we only present the current state of the work, the local deployment.

The contributions of this paper are as follows:

- We present a service model providing services autonomy (each service has control on the logic it encapsulates), reusability (a same service can be reused in different contexts), description, publication and composition abilities. (cf. 3.1)
- We design the dependencies of a given application as a logical dependency graph where nodes are services and components, and edges represent their links. Logical operators between the edges are included, in order to enlarge the choice of services and components. (cf. 4.1)
- We design and implement graph-based decision algorithms for taking loading decisions depending on contextual properties, and adaptation algorithms to adapt the loaded applications to the current runtime context. (cf. 4.1, 4.2, 4.3)
- We show that our approach is efficient with comparison to an existing similar approach through measurements and experiments. (cf. 5)

The remainder of this paper is as follows. In section 2, we present a state of the art of deployment platforms. Section 3, details the design principles. Section 4, presents the architecture of the AxSeL system. Section 5, introduces the AxSeL prototype and the measurements of the conducted experiments. Finally, section 6 concludes and gives the future research works.

2 State of the Art: Software Deployment Systems

Software deployment is defined in [2] as being a process composed of several phases as follows: release, installation, activation, deactivation, adaptation, update, uninstall and retire. Let classify the application deployment approaches using the following criteria, *(i)* the deployment target either distributed or local and *(ii)* the deployment constraints.

Software installers as InstallShied [1] and PC-Install [2] are solutions for software local deployment. Stand-alone applications are packaged into self-installing archives and installed on the terminal. Current installers include Internet extensions for updating new versions and adding new plugins. Though such solutions are efficient to local deployment, they deal with coarse-grained applications and impose their total installation on the terminal, without respecting contextual constraints.

Several research works have been carried out in the area of component-based applications deployment. Contrary to the previous approaches, these solutions target distributed hosts and respects several constraints, in order to provide solutions to the CPP (Component Placement Problem). They are interesting because a decision process is taken at the deployment phase.

[3] considers the network fragmentation issue caused by hosts volatility in pervasive environments. The authors consider a constraint-driven component deployment (installation and activation) in dynamic networks. They deal with a hierarchical model of components which is deployed by propagation over a set of nodes. Resource constraints are expressed through an Architecture Description Language (ADL) that is considered in a decisional algorithm. [4] rely also on a declarative language expressing the constraints and propose a deployment model and an autonomous management of distributed component oriented applications. Constraints as the placement of a component in a node or the components interconnection topology are first described, then resolved in order to find an optimal configuration.

Context constraints may also be related to terminals hardware features. In [5], the authors target the energy economy of the nodes and resolve the CPP through an AI (Artificial Intelligence) approach. The Sikitei algorithm [6] is extended in order to give a component deployment planner. In Smart Deployment Infrastracture (SDI) [7], the authors aim to optimize the memory use in mobile devices. During the component installation, the latter is either installed locally, or called remotely. Both the device memory and the geographical position are considered for selecting the appropriate component of a given application.

User preferences are also considered as contextual information and are used in [8] to provide an adaptation mechanism for service-oriented applications in ubiquitous environments at execution. In order to achieve contextual adaptation, the authors choose at first the services and applications providing a given task. Then, they proceed to components allocation to the selected hosts. Finally,

[1] http://www.installshield.com/
[2] http://www.sharewareconnection.com/titles/pc-install.htm

reconfigurations are possible in case of context changes. An analytical model and an algorithm are proposed to provide the optimal configuration.

System	Deployment	Unit	Decision
[3]	installation and activation	component	ADL
[5]	installation	component	artificial intelligence
[4]	configuration and update	component	deployment descriptor
[8]	configuration and adaptation	service	mathematic model
[7]	installation	component	deployment descriptor

Table 1. A Comparison of Contextual Deployment Platforms

In table 1, we make a comparison of the cited approaches considering three criteria as follows: the deployment phase, unit and decision. The decision criterion refers to the technique the authors adopted to place the given components on the present hosts in the most optimal way according to contextual constraints. As previously mentioned, adopted decisions may result from a logical reasoning on a set of predefined rules, or may follow some static deployment descriptors, or may be mechanisms considering artificial intelligence planners.

Our work deals with service-based applications deployment on local constrained devices by proposing a hybrid (local and distributed) approach. Both issues related to distributed deployment solutions (disconnections, network constraints, services links) and to local deployment (device constraints) are considered. Moreover, we need to be aware of the user preferences and the changes of the surrounding environment, which is similar to the deployment conditions of the previous solutions. Most solutions cope with these problems by executing applications in a degraded mode when deployment conditions are altered (no available hosts, no sufficient resources). AxSeL borrows this technique in order to ensure the application execution even in the worst conditions (luck of resources, disconnection, disappearance of a service). A deployment decision is also taken in order to perform the optimal actions respecting the given conditions.

3 Design Principles: Application and Context

This section briefly outlines the design principles of our platform.

3.1 Application/Service/Component Model

Requirements. The considered applications are composed of services and components. The service/component model needs to satisfy the following requirements.

- **Separation of Concerns [9]**, is the fact of encapsulating software functionality in modular boundary units as services for example. This principle provides several abilities as software reuse, modularity, dependency injection, inversion of control and loose coupling. (1)
- **Description and Publication**, description allows to affect to the services and components contextual descriptions including their features and dependencies. The descriptions may help the decision process by giving information about the deployed modules (services/components). The publication allows the visibility and reuse of services. (2)
- **Resilience**, we need to ensure the application function even if some services disappear or cannot be be loaded. We thought about using special services answering by a null response. These may be used in order to replace the disappeared ones or those that do not respect the deployment constraints. (3)

Definitions. In order to satisfy the cited requirements we refer to definitions present in literature and provide our own definition of the service/component model we use in our approach. [10] present a state of the art of software components from which we cite three definitions.

- *Szyperski*: *a component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* (1)
- *Meyer*: *a component is a software element (modular element), satisfying the following conditions: (i) It can be used by other software elements, its "clients". (ii) It possesses an official usage description, which is sufficient for a client author to use it. (iii) It is not tied to any fixed set of clients.* (2)
- *Heineman and Councill*: *a component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.* (3)

A component is a software unit encapsulating functionality. It can be deployed and executed (definition 1 and 2, requirement separation of concerns). A component has interfaces. An interface corresponds to a service. Services allows the import and export of functionality (requirement 1 (Inversion of control)). A component is subject to composition with other components through published and imported services (definition 1, requirement 1 and 2). A component and its services are not dedicated to a special user (definition 3, requirement 1 (Loose coupling)).

Finally, during services deployment, some services may not respect the deployment constraints or may disappear. Hence, we need to define a null service in order to replace them until the coming of another suitable one. The considered service/component model is depicted in the figure 1.

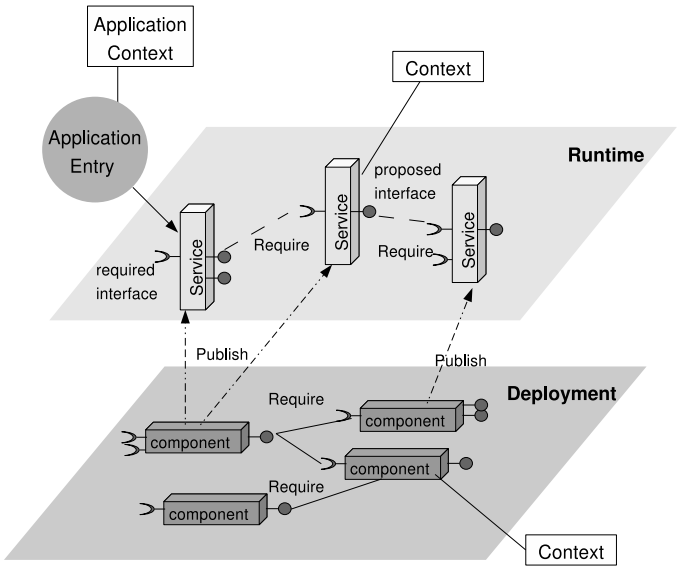

Fig. 1. AxSeL's Application/Service/Component Model

Components dependencies belong to the deployment step, while services dependencies belong to the execution step. Dependencies are satisfied if the needed services are available. Services discovery is made either through a statical way by services and components repositories for example, or through a dynamic way by discovery protocols as UPnP [11], Jini [12], etc. Several platforms relying on the component model exist as EJB [13], CORBA [14], OSGi [15] or web services [16].

We separate services from components in our model, in order to associate to each a description context. We base on these contexts to operate the deployment decision according to the predefined constraints. The loading decision can be made on both the components layer or the services layer. Context models are defined in (cf. 3.2).

3.2 Context Model

Requirements. Our context model needs to satisfy these requirements.

- **Resource-constrained devices**. A description of the current state of the device can be helpful to take the optimal loading decisions. Any noticed change in the device resources may influence the decision process.
- **Application, services and components**. Considered applications are composed of several dependent services. The state of appearance/disappearance of services and their features (name, version, location, etc.) need to be taken into consideration in order to achieve context-awareness. The optimal decision is then taken obeying to the application and services constraints.

– **User**. User may have preferences regarding the applications to be loaded. Security, preference levels can be expressed for the services or applications. These information may also influence the loading decisions.

Definitions. We base on a generic definition of the context and apply it to the above mentioned requirements of our system. In [17] the context is defined as *any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.* We borrow the previous context definition and define our generic context model -depicted in figure 2- as being a set of information gathered from applications, services, components (name, version, needed resources, etc.), devices (free memory, battery, etc.) and users (preferences).

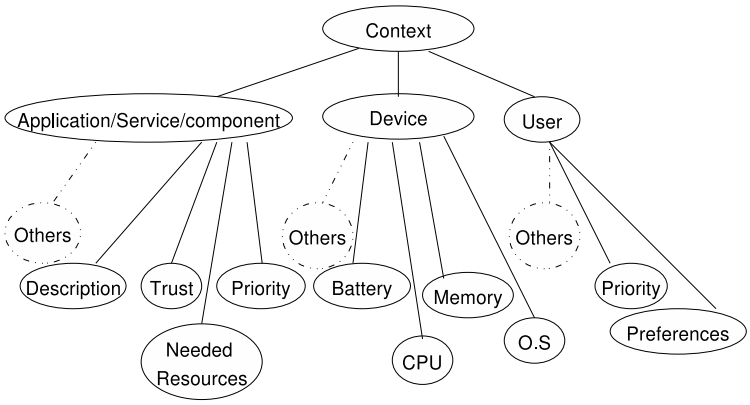

Fig. 2. Context Model

The AxSeL platform uses these contextual information to take the service loading decision by comparing a required context (service, user) to a provided one (device). In this paper we consider a simple instantiation of the above depicted model as being the criteria taken from the device (available memory), from the service (needed memory and assigned priority) and from the user (required priority).

– **Priority**. We assign to each service a priority level expressing its importance for the application. We suppose that the service provider affects this priority to express the necessity of loading or not a given service. The service must be loaded if it has a high priority. Otherwise, it is not necessary to have it on the device.
– **Memory size**. In order to optimize the memory allocation, we introduce the memory size property. It informs about the memory occupation needed by a service in execution. We suppose that this property is provided by the service provider. If the service needs more than the available memory we

don't load it and replace it if possible by a more adequate one consuming less resources.

Nothing prevents from considering other constraints in order to adapt to the users, devices and services requirements. These constraints can be related to the service level (trust, life cycle, etc.), or to the hardware level (other resource needs), or to the user preferences.

4 The AxSeL Architecture

In pervasive environments, services are hosted in mobile devices or remote repositories. Services repositories are described through descriptors including services information (dependencies, location, memory size, etc.). Mobile devices access to the repositories and choose the needed services through the services descriptor. Once the component providing the service is found, it is locally loaded on the device. AxSeL aims to achieve context-awareness by considering a global view of the available repositories. An overview of the AxSeL architecture is illustrated in the figure 3.

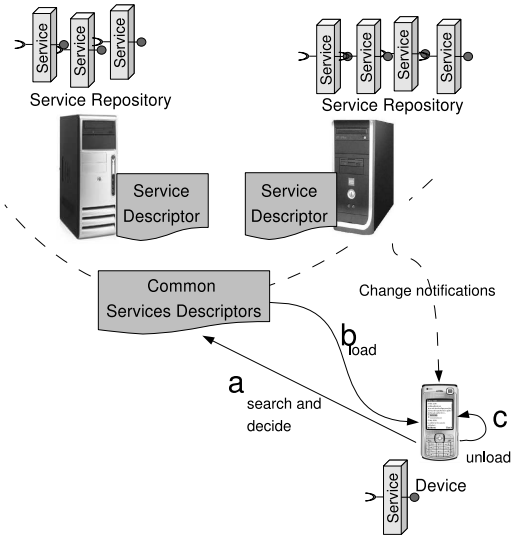

Fig. 3. The behaviour of the AxSeL architecture : (a) the device looks for a service and decides to load it, (b) the service and its dependencies are loaded locally in the device, (c) in the case of changes the service may be unloaded

In order to operate the services loading, the AxSeL architecture follows four steps.

1. **Dependency Extraction**, is the process allowing the services dependency extraction from the services descriptor. After the extraction a service dependency graph is obtained. The graph nodes include services and components properties (unique service descriptor, figure 3),
2. **Loading Decision**, consists in a graph walk with a simultaneous comparison between the required and provided contexts. For each service of the graph a loading decision is taken (step a, figure 3).
3. **Loading**, is the step that applies the previous decision step and operates the effective components loading, installation and execution (step b, figure 3),
4. **Contextual Adaptation**, services and devices state changes may trigger a contextual adaptation as a new service loading/unloading decision, (step c, figure 3).

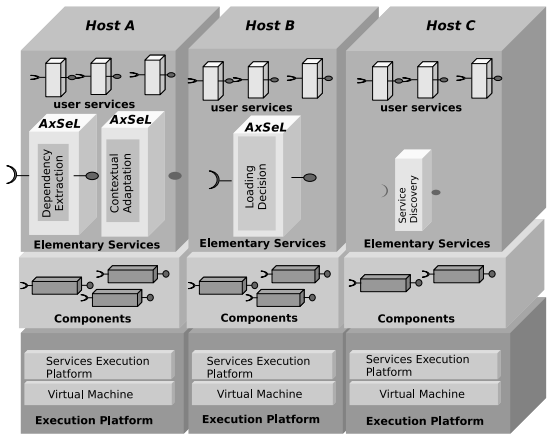

Fig. 4. A layered view of the execution platforms

The AxSeL architecture relies on the service-oriented paradigm, each one of the step 1, 2 and 3 represent a service. Hence, the extraction and adaptation services may be executed in the host A, while the decision service in the host B (figure 4).

In the following sections, we present the service dependency graph, the proposed loading decision, and finally the contextual adaptation. The loading mechanism is realized by the underlying execution platform (Section 5) and will not be detailed in this paper, hence.

4.1 Graph Extraction: Bidimensional Graph Model and Algorithm

Graph Structure, we extract from an XML services descriptor the dependency graph of a given service. The dependency graph includes the services, the components exporting them and their properties (name, location memory size). Services and components properties are included in order to be taken into account

during the decision process. The loading decision may be taken in the service level or the component level, hence the use of a bidimensional graph (figure 5). The latter is a directed graph where services and components are represented by nodes and their dependencies by edges.

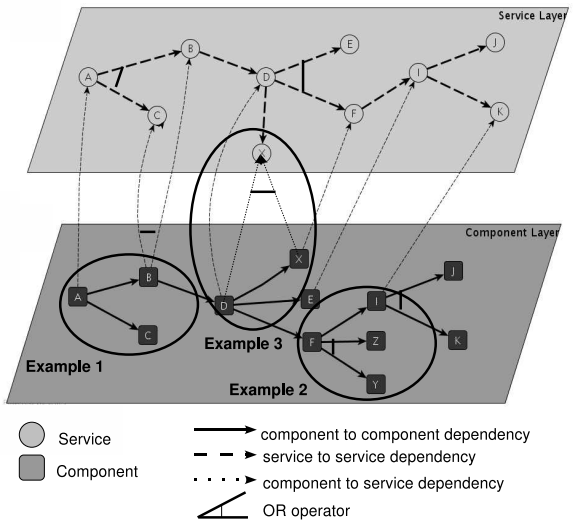

Fig. 5. A Service Dependency Graph

- **Nodes**. The graph nodes represent the services and the components. In the figure, services are represented with round nodes, while components are square shaped. Each node has a weight and some properties.
- **Edges**. They represent services and component dependencies. Nodes that are in the same level are linked by horizontal edges, while nodes from different levels are vertically linked. We map the proposed service/component model (cf. 3.1) to a bidimensional graph. The graph dependencies represent the import/export relationship between the model elements (nodes). Services are implemented by components and components may import other components. Finally, services may import other services. An edge goes from a service to another, or from a component to another or from a component to a service. Edges are directed and may have specific annotations.

Two logical operators are included in the service/model dependency graph, the AND and OR operators. A dependency having the AND operator implies the necessity of loading its nodes. This case is illustrated in the example 1, in the figure 5. The component A must load the components B and C. A dependency having the OR operator represents the possibility of choosing a node or another.

This operator is helpful if many versions of the same service for example are available. We include the OR operator into our model in order to allow the possibility of drawing services from several repositories. In the example 2 of the figure 5, the component F depends from the components I AND (Z OR Y). Finally, the example 3 of the same figure shows the case where a service is published by one or more components. The service X may be provided either by the component X or by the component D.

Graph Extraction Algorithm, two operations contribute to make the extraction process.

1. **Dependency Retrieval**, this operation is based on the service descriptor parsing. The term *Resource* designates a service or a component. The function (*getRequirements()*) helps to extract the Resource dependencies (*Requirement*). After that, for each requirement a recursive dependency extraction is launched. The algorithm 1 takes place until all the dependencies of a service are extracted,
2. **Graph Construction** is made as the previous step is taking place. The nodes are created and included in the wright level according to their types. A service is represented by a node in the service level and a component is represented by a node in the component level. Edges are included between nodes from the same or different levels. At each step of the algorithm, we include in the current graph the dependency graph of its children. The entry point of the current graph is then linked to the entry point of the children dependency graph. The complexity of the extraction algorithm is evaluated to O(nlog(n)).

Algorithm 1 Graph extractDependency(Resource resource)

```
1: G ← new Graph(resource)
2: Requirement[ ] children ← resource.getRequirements()
3: for i, Gi ← extractDependency(children[i])
4: if Gi.firstNode provides Service then
5:     include Gi.firstNode in ServiceLayer
6: else if Gi.firstNode provides Component then
7:     include Gi.firstNode in ComponentLayer
8: end if
9: G.include(Gi)
10: G.addDependencyEdge(G.firstNode, Gi.firstNode)
11: return (G)
```

4.2 Loading Decision: Graph Colouring

Once we have extracted the dependency graph of a given service, we go to the next step which is the loading decision. The loading decision is based on

a comparison between services, users and devices properties. We only load the services obeying to the contextual constraints. The decision process is operated through a graph colouring algorithm.

Colouring Principle, we walk the extracted bidimensional graph and assign to each node a colour according to the taken decision.

- **Red** is assigned to nodes obeying to the loading criteria and that will be loaded on the device.
- **White** is assigned to nodes that are not obeying to the loading constraints. In this case, we create a null service and direct the call to it (cf. 3.1). We notice that a white node cannot break a red nodes series.

During the graph walk, when a red node is encountered it is loaded. When it is a white node, the AxSeL architecture creates a null service and redirects the call to it. The fact of redirecting to a null service prevents the loaded application from a fully work. However, by using this technique we guarantee first, the good work of an application (no crash caused by a missing service). Second, the loading of the needed services only. Once, a suitable service obeying to the deployment constraints appears we replace the null one by the new one.

Colouring Algorithm, this algorithm is inspired from the dedicated algorithms found in the graph theory literature. Several constraint-driven algorithms as Kruskal, PRIM, Dijkstra, Bellman-Ford-Moore, Floyd Warshall [18] has proved their efficiency in extracting the shortest path. Such techniques are helpful to take the loading decision according to constraints as the components sizes and the device memory. Heuristic methods as genetic and A* algorithms are also interesting to optimize the graph walk.

The colouring algorithm of the AxSeL architecture uses two metrics representing the service priority (prio) and memory size (sizeNode).

- **prio()** is a function returning the priority of a service. prioMin is the lowest priority level a service can have to be loaded. The user defines the value of prioMin as a boundary for the services to be loaded,
- **sizeNode** represents the current overall memory size of the red services. sizeMax is a constant value representing the maximal memory size afforded by the device. When sizeNode reaches the value of sizeMax services are no longer loaded.

The graph colouring is made following two steps:

1. **Initial white colouring**, the entire graph is coloured in white at first. By default, none of the services is loaded, but directed to null ones. This step is not resource consuming since we do not really walk the entire dependency graph. At the creation step the graph nodes are already marked by the white colour.
2. **Red colouring**, we walk the dependency graph and assign to the nodes obeying to the loading constraints the red colour. This step is initiated by a call to the colouring algorithm 2 having as parameters the graph entry point and a null value for the variable sizeNode colour(new List[G.firstNode], 0).

Algorithm 2 List colour(List nodeList, Integer sizeNode))

```
1: nodeList.sort(prio)
2: firstNode = nodeList.firstElement()
3: if firstNode.getPrio() ≥ prioMin and sizeNode + firstNode.getSize() ≤ sizeMax
   then
4:    firstNode.changeColour(red)
5:    return colour(nodeList-firstNode+children(firstNode), sizeN-
      ode+firstNode.getSize())
6: else
7:    return colour(nodeList-firstNode,sizeNode)
8: end if
```

First, the algorithm sorts the nodes depending on their priority. Second, it takes the first node of the sorted list and checks if it obeys to the deployment constraints (priority and memory). If it is the case, the node is coloured in red, otherwise, it is kept white. The red node is removed from the initial list and its children are added to be proceeded. The result of this algorithm is a list of coloured nodes. We have evaluated its complexity to O(nlog(n)).

4.3 Dynamic Adaptation: Events and Actions

High mobility in pervasive environments causes many variations that should be taken into consideration. The AxSeL architecture uses a context listener mechanism to capture the context changes such as the device memory state, the user preferences and the services current state. An adapting algorithm answering to the event changes is implemented and presented in this section.

Adaptation Principle, the context dynamics are taken into account in the AxSeL architecture through an event model. Two kinds of events can occur.

- **Service related Events**, this kind of events is triggered when new services appear, or old ones are updated (new version, property changes, disappear, etc.).
- **Device related Events**, these events are triggered in the case of change in the device resources. For example, if the device memory is released it can host new services.

Both events trigger new actions on the graph.

- **Graph Structure Modification**, the service appearance or disappearance causes respectively a node addition or removal in the dependency graph, and the related dependencies.
- **Node Colour Modification**, the state of the device memory influences the state of the nodes. Actually, if the memory is freed, loading new services becomes possible (red colouring). Besides, if the memory is full some nodes may be removed (white colouring). The user preferences also influence the state of the nodes. That is, when the user needs to load/unload a

node or to modify its priority. Finally, when the state of a service changes (appearance/disappearance/modification) it changes the colour of the node representing it.

The cited actions initiate a new decision process. However, the colouring algorithm is incremental and takes into account the services already loaded without making a new graph walk and re-colouring.

Adaptation Algorithm, when an event is notified the algorithm 3 is initiated. Events emanate from two possible sources.

- **Node Addition**, we check if the new node has a red parent in the graph. In this case, we initiate the colouring process by adding the new node and updating the sizeNode variable.
- **Node State Change**, if the node is already red we uncolour it (white) and reduce its memory size from the sizeNode variable. Finally, we uncolour its children (algorithm 4). The nodes recolouring is initiated with the input variable sizeNode.

Algorithm 3 adaptation(Event e)

```
1: if e.getType() == NewNode then
2:    if ∃ nodei / father(node) and nodei.isColored(red) then
3:       colour(l+node, sizeNode)
4:    end if
5: else if e.getType()== NodeChange then
6:    if node.isColored(red) then
7:       node.changeColour(white)
8:       sizeNode ← sizeNode - node.getSize()
9:       uncolour(new List (children(node)))
10:      colour(l + node, sizeNode)
11:   else if node.isColored(white) and ∃ nodei / father(node) and
      nodei.isColored(red) then
12:      colour(l+node, sizeNode)
13:   end if
14: end if
```

The uncolour(List) function checks if a red node has not red parents. In this case it is possible to uncolour it (white). The node memory size is deduced from the variable sizeNode and the function propagates the uncolouring process to the children. A processed node is not considered again. We evaluate the complexity of this algorithm to $O(n^2)$ because the uncolouring process re-walk the entire dependency graph to retrieve the node parents.

Algorithm 4 uncolour(List l)

```
1: ∀ node ∈ l
2: if node.isColored(red) and ∄ nodei / father(node) and nodei.isColored(red) then
3:     node.changeColor(white)
4:     sizeNode ← sizeNode - node.getSize()
5:     uncolour(l - node+ children(node))
6: end if
```

5 Implementation and Evaluation

The AxSeL architecture is implemented as a java based prototype. It relies on the java virtual machine and the Felix [19] platforms. Felix is an OSGi [15] specification platform dedicated for the services deployment and execution (figure 4). We match the service/component model with the OSGi platform. A component is implemented as an OSGi bundle (deployment unit composed of classes and a manifest). A service is implemented as an OSGi service, it is represented using a java interface.

Measurements have been performed to prove the feasibility and efficiency of the AxSeL implementation. In order to represent a use case application, we have examined the available Felix repositories. Applications in the Felix repository have an average number of services about twenty services. Hence, we have performed tests using a dependency graph of services representing an application composed of twenty. Each node may have one or five dependencies at the component level, and only three dependencies at the service one. Service providers and descriptors may give wrong or unchecked services descriptions representing cyclic dependency graphs. AxSeL takes into account this criterion and performs the services loading even in case of cyclic graphs without altering its performances. Experiments are made according to the phases of graph extraction, loading decision, and dynamic adaptation. We evaluate the algorithms execution time (ms) and memory occupation (Ko) complexity in two cases. First, in case of increasing the graph nodes. Second, in comparison with an existing platform.

Time/Memory Measurements Time and memory cost of each phase (algorithm) is evaluated by increasing the number of the graph nodes. Figures 6 and 7 depict the curves of each algorithm.

Observations, The adaptation phase is tested in the worst case that is the entire graph uncolouring and recolouring. The figure 6 shows that AxSeL has a quite linear increasing tendency at the extraction phase. The decision curve is linear and almost null even in the largest dependency graph (not visible in the figure). Finally, the adaptation curve is pertaining to the exponent.

Figure 7 shows the memory use cost in response to the graph nodes number variation. At the extraction phase, the curve follows an asymptotic behaviour. At the decision phase, AxSeL have a constant memory use cost. Finally, the adaptation curve of AxSeL has an oscillatory trend.

Analysis, time and memory measurements show costs tendencies that are equivalent to the previously evaluated theoretical complexity of each algorithm. Though the adaptation phase has to be improved, it does not really alter the performances of AxSeL, since the adaptation is tested in the worst case (total graph recolouring) and takes less time and uses less memory in common cases (partial graph recolouring). Its oscillatory trend is explained by the memory de-allocation due to the java virtual machine garbage collector. Its exponential time cost strongly depends on the graph structure and may be improved by optimizing the graph walk through a more efficient parents search method, for example by keeping a list of the dependent nodes at each node. Finally, the gathered results show that AxSeL has a short execution time even in the largest graph (500 nodes) and has an economic memory use. We attribute the latter to the use of the **Singleton** design pattern which allows the reuse of the created objects without re-allocating them in memory.

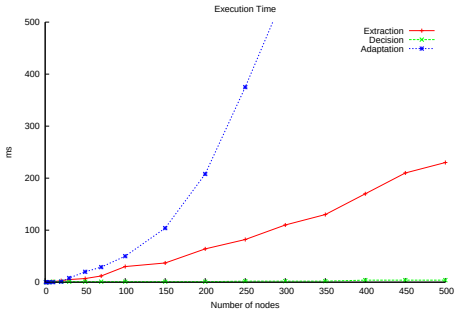

Fig. 6. Execution Time per Phase

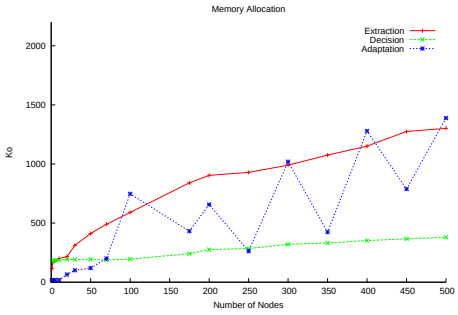

Fig. 7. Memory Allocation per Phase

Comparison with an existing platform The OBR [20] platform installs the OSGi bundles from a remote repository without applying any deployment decisions, neither respecting contextual constraints. It only loads all the needed bundles without making any contextual adaptation. The performed tests were undergone by using a non cyclic graph in order to allow the comparison with the OBR platform that do not take into consideration such cases.

Observations, both architectures are compared during the different phases. In the graph extraction phase (figure 8), AxSeL takes on average twice less time than OBR. The decision graph colouring algorithm is ten times faster in AxSeL than in OBR. AxSeL has an efficient contextual adaptation feature which is absent from the OBR platform. Figure 9 shows the trend of the observed memory allocation. At the graph extraction phase, AxSeL uses more memory resources than OBR. However, for the compared phases AxSeL is on average, over twice less memory consuming than OBR without the adaptation phase, and one time and a half considering the former.

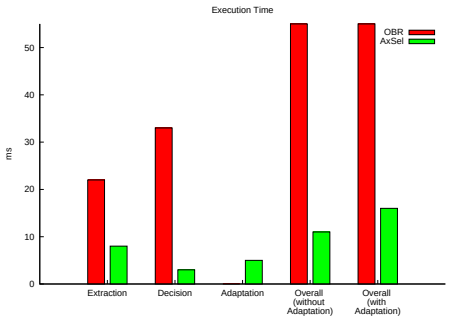

Fig. 8. Execution Time

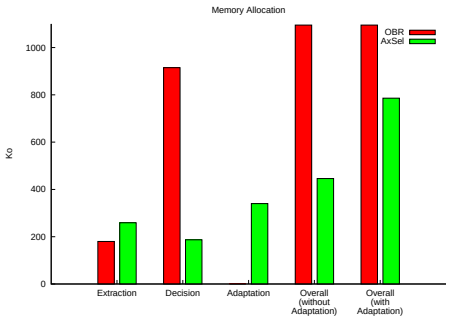

Fig. 9. Memory Allocation

Analysis, AxSeL is faster in deciding which nodes to load because it proceeds to a unique graph walk, while OBR makes several walks to look for the nodes dependencies. Moreover, at the extraction phase, AxSeL uses more memory than OBR. Actually, AxSeL extracts a bidimensional graph, while OBR do not consider graph structures. AxSeL is efficient in comparison with OBR in execution time and in memory allocation. However, the current version of OBR can largely be improved.

6 Conclusion and future works

In this paper, we have presented an architecture for contextual services loading: AxSeL. The key design strategy of our approach is in using the service-oriented applications paradigm and applying inspired graph theory algorithms in local applications deployment. Services dependencies are represented in a bidimensional graph. The graph nodes are enriched with the services and components contextual properties as the memory size and the priority. The graph is extracted from a common services descriptor where the services dependencies are presented. The loading decision is then taken through a graph colouring algorithm. The loading decision is expressed by two colours *red* (loading) and *white* (no loading). In the case of the non loading of a service, we direct it to a null service in order to ensure the application resilience. The contextual adaptation is realized through mechanisms of listening, capturing and interpreting context events. Our approach supports both the initial deployment as well as the later reconfiguration of services. We have designed, implemented and evaluated the AxSeL architecture that is composed of services able to be deployed on a distributed environment. This avoids the use of the device resources by greedy processes of graph walks. Each algorithm (extraction, colouring, adaptation) is available in an isolated service. Performed measurements and comparison show AxSeL's efficiency in memory use and execution time.

In the near future, we are going to establish several dynamic deployment strategies according to the contextual constraints. This will increase AxSeL's

autonomous adaptability and is possible through the Strategy design pattern that is already used but not fully exploited. Moreover, we plan to introduce the remote services use in our platform, in order to make it really hybrid (local and remote). Finally, there is an opportunity to develop more elaborate quantitative comparison, potentially based on different decision algorithms and conducted in different contexts in order to evaluate AxSeL's adaptability and its resource measurements.

References

1. U.Zdun, C.Hentrich, Aaalst, W.M.D.: A survey of patterns for service-oriented architectures. In: International journal of Internet protocol technology. Volume 1. (2006) 132–143
2. Carzaniga, A., Fuggetta, A., Hall, R.S., Heimbigner, D., Hoek, A., Wolf, A.L.: "a characterization framework for software deployment technologies". Technical report, CU-CS-857-98, University of Colorado (1998)
3. Hoareau, D., Mahéo, Y.: Middleware support for the deployment of ubiquitous software components. In: PUC conference. (2006)
4. Dearle, A., Kirby, G., McCarthy, A.: A Framework for Constraint-based Deployment and Autonomic Management of Distributed Applications. In: ICAC conference. (2004)
5. Kichkaylo, T., Karamcheti, V.: Optimal Resource-Aware Deployment Planning for Component-based Distributed Applications. In: 13th IEEE ISHPDC. (2004)
6. Kichkaylo, T., Ivan, A., Karamcheti, V.: "Sekitei: An AI planner for Constrained Component Deployment in Wide-Area Networks" tr2004-851. Technical report (2004)
7. Taconet, C., Putycz, E., Bernard, G.: Context-Aware Deployment for Mobile Users. In: 27th IEEE ICSAC conference. (2003)
8. Poladian, V., Sousa, J., Garlan, D., Shaw, M.: Dynamic configuration of resource-aware services. In: 26th ICSE conference. (2004)
9. Mili, H., Elkharraz, A., Mcheick, H.: Understanding separation of concerns. In: EA workshop. (2004)
10. Kui, K., Wang, Z.: Software component models. IEEE TSE conference (2007)
11. Microsoft Corporation: "Understanding UPnP: A white paper". Technical report, UPnP Forum (2000)
12. Kumaran, S.: Jini technology an overview. In: Prentice Hall PTR. (2002)
13. Hamilton, G.: The javabeans specification. In: Sun Microsystems. (1997)
14. Zahavi, R.: Entreprise application integration with corba component and web-based solutions. In: Johne Wiley & sons. (1999)
15. OSGi Alliance: Osgi-the dynamic module system for java. In: http://www.osgi.org/. (2008)
16. Iverson, W.: "Real Web services". O'Reilly (2004)
17. Dey, A., Salber, D., Abowd, G.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. In: HCI conference. (2001)
18. Lacomme, P., Prins, C., Sevaux, M.: "Algorithmes de graphes". Eyrolles (2003)
19. Felix: The Apache Felix Project. In: http://cwiki.apache.org/FELIX/index.html. (2008)
20. OBR: Obr Bundle Repository. In: http://www.osgi.org/Repository/HomePage. (2008)