Service Group Management facilitated by DSL driven Policies in embedded Middleware

Christopher Foley, Gemma Power, Leigh Griffin, Chen Chen, Niall Donnelly and Eamonn de Leastar
Telecommunications Software and Systems Group
Waterford Institute of Technology
Waterford, Ireland
{ccfoley, gpower, lgriffin, cchen, ndonnelly, edeleastar} @tssg.org

***Abstract*— Middleware by its very nature is fundamental to the functioning of systems as it provides the communication between software components. It is very much an underlying technology and is rarely visible to end users. As systems develop, certain domain semantics, provided by the domain experts, need to be injected into the behaviour of the underlying middleware, but in a controlled manner. The methods used to achieve this are often static in nature, wholly dependent on how they are implemented, deployed and managed. An increasingly popular way to manage this behaviour injection is through the use of policies, a technique used to govern defined rules, triggered by associated events, resulting in specific actions when certain conditions are encountered. Strong efforts have been made throughout the evolution of software development methods and programming languages to solve the lack of dynamicity which can arise through poor practices. Successive language based attempts to attain a higher level of abstraction in the notations used and techniques deployed have resulted in the re-discovery of Domain Specific Languages (DSL). This paper looks at injecting the dynamicity required in the management of service groups through a policy based DSL.**

Keywords- Middleware; Policy Engine; Domain Specific Language; Services; Group Communication,

I. INTRODUCTION

The middleware created as part of the IST MORE project [1] is a Service Orientated (SOA) [2] middleware targeting the embedded device environment. One of the functional utilities it provides to developers is group communication. The middleware allows the formation of groups of services and the distribution of messages between the group members. The reason for providing service group functionality is that one middleware service, by itself, provides less scope when solving problems. A group of services, each with different tasks, working together, gives greater capabilities when developing solutions for larger more complex problems. Managing and administering these groups of services, steered by domain semantics, then becomes a necessity. In order to achieve a valid solution to this, a number of issues must be addressed:

- Introduce a mechanism to govern the service groups
- Bridge the gap between the domain expert (with little or no middleware expertise) with the middleware services, which will allow the insertion of the domain semantics.
- Add dynamicity to this governance of these service groups
- Perform all of the above in the embedded environment. i.e. small memory footprint

A lightweight rules based system was decided upon to govern and manage the service groups. The design of the key components supporting this system would address the issues raised above. Those components, a lightweight Domain Specific Language (DSL) [3] and the policy processing middleware service, are freely available from [4], along with the rest of the software developed.

This paper is broken down into seven sections. This introduction serves as the first. Section two examines the technological choices made to support the system. The third section looks at related work. Section four, the architecture section, examines the underlying architectural components developed and deployed. The fifth section focuses on DSL Group Policies. Section six documents the Testbed created for validation. The seventh and concluding section also examines the Future Work to be carried out.

II. TECHNOLOGY (DSL/GROOVY)

In Object Oriented approaches to programming, there is a movement to attempt to have a strong representation from problem domain entities within the solution space. Termed Domain Driven Design [5], this has the benefit of enabling domain experts validate a design, ensure its consistency and be sufficiently well informed on the emerging solution to contribute meaningfully to feature evaluation and ongoing trade-off decisions.

This Domain Driven Design movement has largely relied on modelling notations and analysis patterns as the shared vocabulary between domain and solution experts. An overlapping and equivalent movement from a language perspective is the recent re-discovery of Domain Specific Languages (DSL) [6]. Here the emphasis is specifically on the programming language itself and on devising a language that can directly represent domain-oriented concepts and techniques. With the language very much centre stage the emphasis moves from modelling to implementation. So, in solving a problem, a complimentary DSL is selected, or often devised, to more closely suit the problem domain [7]. The notation of the solution is then potentially capable not necessarily of being written by domain experts, but at least it can be read and understood by them. This offers obvious

benefits in verifying correctness, maintenance and overall flexibility and accuracy of the solution.

There has also been considerable recent interest in how to engineer the DSL. In many ways this is a well understood technology - ANTLR being a prominent current tool [8] - and relies on modern incarnations of traditional compiler technology. However, Fowler has coined the term *internal DSL* [9], which contrasts with this approach (which he terms *external DSLs*). With an internal DSL, we rely on the flexible nature of the programming language itself. This flexibility enables idioms and patterns in the host language to facilitate a fluency of expression that can be very convincing in the context of a specific domain. Thus we can invent what amounts to a *dialect* of the host language which targets a specific problem. This dialect is not translated into the host, or any other language. It is merely an adaptation of the language along a particular axis, delivering a notation a domain expert could conceivable read with ease.

While there have been attempts to compose DSLs in Java [10], the statically typed nature of the language can be a limitation, enabling what can be termed a more *fluent* set of expressions, if not quite a full DSL. Java compatibility is attractive though, and it is possible to dovetail Java with DSL capabilities. This is most compellingly done if we can employ a dynamically typed language that is also java compatible. Groovy is one of the most prominent of this breed, and is a hotbed of DSL experimentation and innovation [11]. Although not exclusive to Groovy by any means, key language mechanisms such as closures, dynamically bound scripts, builder pattern, operator overloading, and meta programming capabilities enable highly expressive and domain focussed dialects to be embedded directly into the language, without the need for complex grammar or translation phases.

III. Related Work

A major requirement for the MORE project was the ability to dynamically manage groups of services in an embedded environment. While investigating the use of a policy engine several existing policy engines were evaluated. These include [12] [13] [14] [15], the reason these applications were deemed unsuitable was largely due to their memory footprint and complexity. For example while [14] actually has a small footprint it has a significant working memory requirement, unsuited to the embedded domain.

Some other projects investigating a similar alternative include [16] which published an article highlighting the possible advantages of using Groovy to create a DSL to build a policy engine. The use of policies associated with a service group, inspired by Policy Based Network Management was described by the authors in [17]. [18] discusses a framework for flexible composed service charging. The scheme was developed using a Groovy-based DSL which allowed end-users to change their charging rules, with the modification rules reflecting the business relationships between different service providers and their customers. Our Policy Engine differs from [18] in that the policies can be dynamically reconfigured, published and executed in real time. Previous Policy Engines based on DSLs generally pertain specifically to Security Policies, [19] [20] or Network Management [21].

No suitable comprehensive policy based solution appropriate for the embedded domain was freely available. As such it was decided to develop one.

IV. Architecture

A. MORE Middleware

The middleware which is central to the solution is based on the work conducted during the IST-MORE Project funded by the European Union in the 6th Framework Programme. MORE is a cross-platform and service oriented middleware for distributed communication systems allowing for dynamic service deployment in pervasive environments. MORE enables efficient service development for devices like smartphones, mid-sized embedded systems and normal PCs. The MORE middleware architecture is based on the Service Oriented Architecture (SOA) approach.

The MORE middleware is realised through sets of enabling services. The most basic deployment will consist of the MORE CORE, which is mandatory for the middleware to function and contains a minimal set of utility services. The MORE middleware provides optional functionality through these utility services. Examples include Group Management Service, Policy Engine, Security and Compression. The middleware provides the scope for application developers to develop their own enabling services which can then make use of the core and utility functionality provided.

The runtime of the MORE middleware is based on the Device Profile for Web Services specification [22]. DPWS identifies a minimal set of Web Service specifications tailored towards the needs and capabilities of embedded devices in order to allow for a base level of interoperability between devices and standard Web Services.

In conjunction with DPWS, MORE makes use of the OSGi [23] platform for managing the MORE middleware and the user services. OSGi technology provides a service-oriented, component-based environment for developers and offers standardised ways to manage the software lifecycle.

B. Policy Engine & Group Management Services

The MORE Group Management Service (GMS) handles the administration of service groups; group creation, deletion of groups, and the addition of services to groups. GMS also handles the forwarding of group messages to all group members. When a group is created a governing policy can be assigned to that group or not. This policy defines how this group is governed, e.g. if a specific message is sent to the group then a new service of a certain type should be added as a result.

A Policy Editor which encapsulates the DSL has been developed which aids in the writing and deployment of group policies. A policy can be exported to the Policy Engine Service (PES). The editor discovers the running PES and exports the policy to it.

The Policy Engine Service is a standalone service which handles the policy processing based on events supplied to it. Within the MORE context it is the GMS which provides the PES with the events. When a message is sent to the group via GMS, it will be forwarded to the PES if there is a policy associated with that group in question. The PES will treat the incoming message as an Event and check if there are Policy Rules associated with it. If rules exist, they will be processed. If the rules condition evaluates to true, then the GMS will be informed by the PES of the subsequent actions to take. These actions may take the form of group administration actions which will be performed by the GMS.

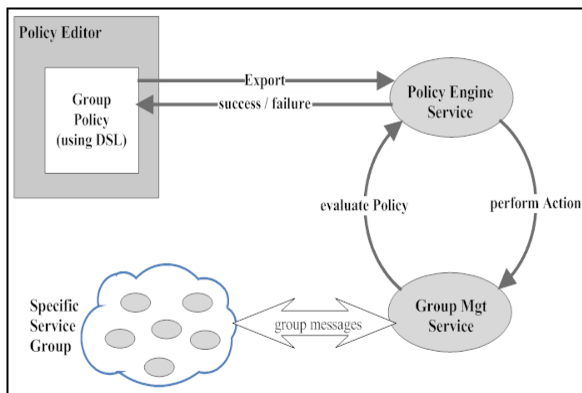

Figure 1. Component Architecture View

Failures within the Policy Engine are addressed by decoupling the Policy Engine from the other system components and by relevant events.

1. The Policy Editor performs syntax checking on each policy before exporting. On export from the policy editor an acknowledgement is received in one of the following forms:
   - Connection failure - if a connection cannot be established with the Policy Engine
   - Syntax failure - if the policy to be exported is not of the correct syntax
   - Successful - if the Policy Engine received the policy
2. Group Management Service message forwarding. The GMS will attempt to invoke the Policy Engine service if there is a policy associated with the group message to be processed. If this invocation fails in any way then GMS will still forward the message to all group members and log an error message indicating the failure to process the policy.

Further work is needed with respect to failover of the Policy Engine, primarily with respect to ensuring the service stays up. A number of methods are being investigated.

V. DSL Group Policies

A. *DSL Policy Scripts and Policy Editor*

The end user, being an expert in a specific area, for example a medical doctor, may not have sufficient programming knowledge to interpret a policy. Therefore, the user will need an interface to the policy engine that is centered on the problem domain, rather than the programming complexity needed to solve it. The policy scripts that are used by the end user to build a policy are based on a domain specific language realised through Groovy. A policy contains a set of rules, with each rule comprised of a collection of associated events, conditions and actions.

As part of the MORE project, a diabetes scenario was documented [24] and will be used here as our policy example. A diabetologist wants to monitor the glucose level of a diabetes patient. This patient has a glucose monitor attached to them, which sends periodic measurements of the patients blood sugar level to the patients care group. The diabetologist only requires to be alerted if the glucose level of the patient exceeds 8mmol/L or if the level falls below 3mmol/L, as shown in Figure 2.

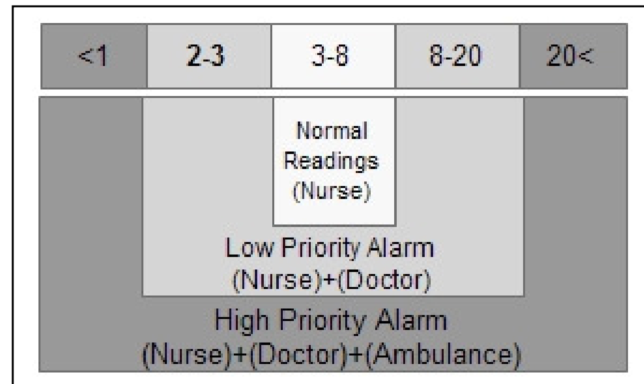

Figure 2. Blood Sugar Range

The Policy Editor is a tool used to facilitate users in the modification and creation of policy scripts as well as dynamic script deployment to the policy engine service. The editor has been developed based on the Eclipse plug-in Architecture [25]. The main features of the editor plugin include content assistant and colour context to aid the end user in the authoring of the policy. The plugin also features a smart export function to discover any running PES and subsequently deploy their new script dynamically. Figure 3 shows a screenshot of the plugin running within the Eclipse IDE. Based on the data from figure 2, the syntax of a sample user policy created with this plugin can be seen in figure 4.

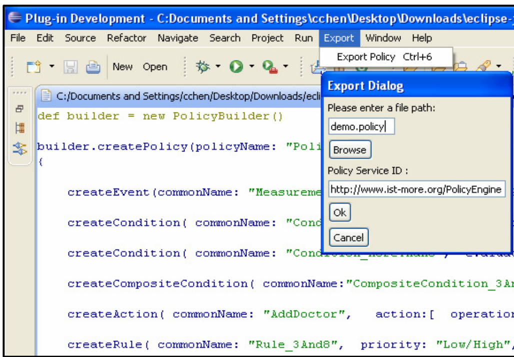

Figure 3. Eclipse Plugin

```
createEvent(
 commonName: "Measurement_Event",
 identifier: "ReceiveGlucoseMeasurement",

parameters:["BloodSugarMeasurementValue"])

createCondition(
 commonName: "Condition_LessThan3",
 evaluation: [
  var :
   eventValue("Measurement_Event",
              "BloodSugarMeasurementValue"
       ),
  op: "lt",
  val: "3" ] )

createCondition(
 commonName: "Condition_MoreThan8",
 evaluation: [
  var :
   eventValue("Measurement_Event",
              "BloodSugarMeasurementValue"
       ),
  op: "gt",
  val: "8"  ] )

createAction(
 commonName: "AddDoctor",
  action:[
  operation: "addMember",
  groupName: "Doctor",
  variable:  "Michael",
  valueParam:"Diabetologist" ])

createRule(
 commonName: "Rule_8",
 priority: "Low/High",
 event: ["Measurement_Event"],
 conditions: ["Condition_MoreThan8"],
 actions: [AddDoctor"])
```

Figure 4. Sample User Policy

B. Policy Builder

A Policy Builder is used to structure the DSL and to generate an “event-condition-action” policy. It is utilised for the interaction of the policy engine and DSL (policy script).

A policy script building approach to policy engine development allows for this interaction as the builder itself can be designed for whatever application is needed in the problem domain (see Figure 5). The user needs to write a DSL to capture their specific domain requirement. Take this example user requirement and its associated DSL interpretation; “the group size of service group A is to be no larger than ten and have at least one doctor as a member”. builder.createGroup (GroupName A,GroupSize 10,Number ofDoctors 1). The builder then takes the attributes set by the user and converts them to code scripts that can be used by the policy engine based on the requirement that the user defines. This code is dictated by the technologies used by the policy engine and not tied to a specific implementation.

The other objective of the policy building approach is to allow for a more advanced approach for development of DSL based scripts. The builders design was architected with a view to future-proofing and extensibility. Therefore, users will be able to update the builder through customisable parameters, taking the form of closures which are passed into the builder for evaluation. The policy builder is initialised in a policy script according to a DSL. Method calls on the builder are intercepted and passed into the policy builder as invoked on the Node Builder [26]. The node is designed to decide what methods to use based on the method call to the policy builder. The builder chooses the method to call (e.g. createCondition() method) and passes in the attributes of the rule from the policy script written by the user. The implementation method takes the attributes and builds Groovy scripts to process the data according to the format the data is coming in as. These closures are then added as attributes to the required entity (event, condition, action). Rules are then assigned to a policy to be used by the policy engine.

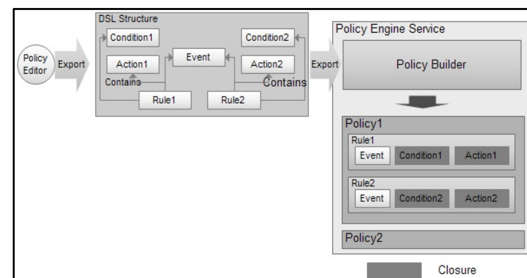

Figure 5. Policy Building Approach

VI. TESTBED

The MORE services were comprehensively evaluated using the testbed configuration depicted below in figure 5

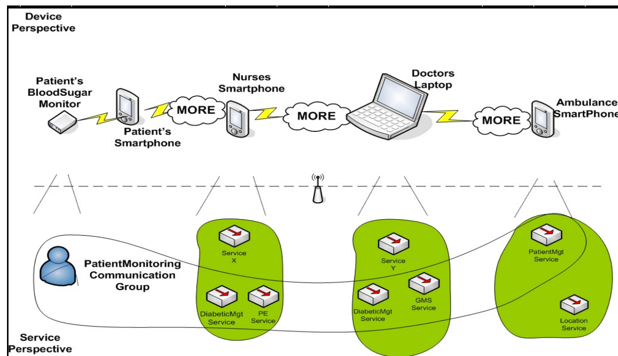

Figure 6. Testbed Configuration

These services needed to be lightweight enough to run on resource constrained devices, as such the Openmoko Smartphone [27] was an integral component of the testbed. This device has a 400 MHz processor and 128MB of RAM with Linux kernel 2.2.24. The Testbed was comprised of an off the shelf blood sugar monitor with Bluetooth capability, a wireless access point configured to provide connectivity, three Openmoko Smartphones and one laptop (tested with both Windows and Linux distributions). Consequently a variety of hardware devices and software components were utilised for effective testing of the Policy Engine Service. The devices and services were chosen to fulfil disparate application domains. As part of the MORE project, two end user scenarios were examined, remote monitoring for healthcare and mitigation management in forestry.

The software components include a subset of the MORE services. The test configuration shown in figure 6 depicts the equipment used in validating the MORE health care scenario which is described throughout this paper, specifically the GMS, PES, Patient Service, Nurse Service, Doctor Service and Ambulance Service. The blood sugar ranges for alerting these parties can be clearly seen in Figure 2. [28] conducted a similar study with respect to remote monitoring and care for cardiology patients. The software components and devices are also visible in Figure 6.

In Figure 6 the Patient Monitoring Group is comprised of, the Patient, Nurse, Doctor and an Ambulance. As inferred from Figure 2, this is an emergency scenario and the Patient's blood sugar level is in the emergency range (below 1 or above 20). A less serious blood glucose reading for example "2" and the policy would dictate that only the Doctor and the Nurse be added to the group to monitor the Patient.

This Testbed configuration was also used to verify the dynamic creation of a policy using the Eclipse IDE and the automatic deployment of this policy to the policy engine service running on the Nurses' device for evaluating new conditions and acting on them accordingly.

The Policy Engine developed for the MORE project can be used as a stand alone product for use in any application. As shown, [28] examined it for use with Cardiology patients. Rich user guides, including video tutorials on policy creation and deployment are available from the website [1]. The code base from the MORE project has also been released as an open source project and is freely available from [4]. Aspects of the PE are already being evaluated for other EU Projects [29] [30].

VII. CONCLUSION AND FUTURE WORK

The MORE project has formally ended and the software produced and discussed within this paper is freely available as open source. As such, the future work is in the hands of the community, however, the authors recommend three potential extension points which will now be discussed. The current service selection model is based on a FIFO style queue for ease of use, as this was not the focus of the research carried out. An intelligent selection mechanism should add services more appropriate to the current requirements and the past history of service interactions within the group. Therefore, the authors recommend that an intelligent service is injected into each group managed by the GMS. The responsibilities of such a service would include monitoring messages and requests sent within the group and between the GMS and PE services. Over time such a monitoring service could build up a historical view of the group and allow the GMS make more informed decisions about service selection.

Another potential extension point could be the autonomous management of the policy engine rule base and associated conditions. The policy rules specified might initially govern the general behavior of services. Specific instances which operate outside of these generic boundaries could be accommodated far easier if changes could be made by the PE service without the need for direct intervention by the domain expert. External influence, be it from a historical group service, as described above, or from a data mining service, abstracting contextual information from devices, could provide the necessary rich semantic information required to make an informed alteration to a live rule.

The final proposed extension point centers on the resiliency of the system to service failure. Ensuring that all available services, particularly those with complex dependencies, remain accessible after a previous failure would be a key requirement. The authors therefore recommend improvements to the current Policy Engine with respect to failover. The design of the policy engine has allowed for failures to be handled as discussed earlier but the work involved for fully securing this area was beyond the time restricted scope of the tasked research. Provisions were afforded for this work in the modular design of the system.

Interpreting a technical vocabulary, already imbued with architectural semantics can be a daunting experience for a domain expert. Expressing the solution space as a readable DSL provides an important bridge between the domain

expert and the system. An added benefit of developing a customised DSL is the lightweight nature of the design, making it an ideal candidate language for deployment within the problem domain. Engineering the core components of this system to be standalone brings flexibility, scalability and independence. This separation of concerns allows the developer, and indeed the domain expert to focus on the task at hand. This paper has proposed such a lightweight rules based system for the dynamic management of group services.

VIII. Acknowledgment

The work described in this paper was partly funded by the European Union through the MORE project in the 6th Framework Programme under the Reference-ID FP6-IST-032939. The authors would also like to acknowledge funding support from the Irish HEA PRTLI Cycle 4 FutureComm (http://futurecomm.tssg.org) programme.

References

[1] IST-MORE [online]. Available from http://www.ist-more.org/ Accessed on 08-JAN-2010.

[2] SOA Service Orientated Architecture[online].Available from http://www-01.ibm.com/software/solutions/soa/ Accessed on 22-DEC-2009

[3] DomainSpecificLanguage [online] Available from http://www.martinfowler.com/bliki/DomainSpecificLanguage.html Accessed on 22-DEC-2009.

[4] Open Source MORE [online] Available from http://sourceforge.net/projects/mores/ Accessed on 15-JAN-2010

[5] Evans, E., Domain-Driven Design - Tackling Complexity in the Heart of Software, 2004, Addison-Wesley

[6] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. ACM Computing Surveys, 37(4):316–344, 2005

[7] Michael Swaine, Language Workbenches: Is This the Era of the DSL?, Pragpub, October 2009, The Pragmatic Programmers

[8] Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, Pragmatic Programmers, 2007

[9] Fowler, Martin, Language Workbenches: The Killer-App for Domain Specific Languages?

[10] Subramaniam, DSLs in Java, JavaWord, http://www.javaworld.com/javaworld/jw-06-2008/jw-06-dsls-in-java-1.html last accessed January 11, 2010

[11]] König, King, Laforge, Skeet, Groovy In Action, Second Edition, Manning, 2009

[12] Drools Policy Engine [online] Available from http://www.jboss.com/products/rules Accessed on 10-JAN-2010.

[13] Hammurapi Policy Engine [online] Available from http://www.hammurapi.com/dokuwiki/doku.php Accessed on 16-JAN-2010

[14] Jess Policy Engine [online] Available from http://www.jessrules.com/jess/docs/index.shtml Accessed on 15-JAN-2010

[15] Pyke Policy Engine [online] Available from http://pyke.sourceforge.net/ Accessed on 15-JAN-2010

[16] Isocra; [online] Available from hthttp://www.isocra.com/2008/01/groovy-dsls-and-rules-engines/ Accessed on 15-JAN-2010

[17] Foley et al, "Distributed Pervasive Services using Group Service communication supporting Body Area Networks" Proceedings of BodyNets 2008.

[18] Jennings et al, "Specifying Flexible Charging Rules for Composable Services", in Proc. 2008 IEEE Congress on Services. pp 376-383

[19] Hamdi et al, "A DLS Framework for Policy-based Security of Distributed Systems", in Proc. 2009 IEEE International Conferences on Secure Software integration and Reliability Improvement. pp 150-158

[20] Nielson et al, "A domain-specific programming language for secure multiparty computation", Proceedings of the 2007 workshop on Programming languages and analysis for security. Pp21-30

[21] Barrett et al, "A Model Based Approach for Policy Tool Generation and Policy Analysis," Proc. 1st IEEE Int'l. Global Information Infrastructure Symp. (GIIS 2007), IEEE, 2007, pp. 99-106. (2007)

[22] S. Chan et al., "Device Profile for Web Services", February 2006 [online] Available from http://schemas.xmlsoap.org/ws/2006/02/devprof/ Accessed 18-DEC-2009.

[23] OSGi Alliance, "OSGi Service Platform" [online] Available from http://www2.osgi.org/Specifications/HomePage, Accessed 18-DEC-2009.

[24] Specification of Validation Services / Scenarios [online] Available from http://www.ist-more.org/images/stories/d3.2_validationscenarios.pdf , pages 36-91 Accessed on 15-JAN-2010

[25] Eclipse Plug-in Architecture [online] Available from http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html Accessed on 15-JAN-2010

[26] Node Builder [online] Available from http://groovy.codehaus.org/GroovyMarkup Accessed on 15-JAN-2010

[27] OpenMoko [online] Available from www.openmoko.org Accessed on 15-JAN-2010

[28] Power et al "An Adaptive Middleware Applied to the Ad-hoc Nature of Cardiac Health Care" Proceedings of MobiQuitous 2008.

[29] PERIMETER [online] Available from http://www.ict-perimeter.eu/index.php Accessed on 16-JAN-2010

[30] EFIPSANS [online] Available from http://www.efipsans.org/ Accessed on 16-JAN-2010.