UbiAcme: Uma Linguagem de Descrição Arquitetural para Sistemas Ubíquos

Carlos Machado[1,2], Eduardo Silva[2], Thais Batista[2], Jair Leite[2], Flavia C. Delicato[3], Paulo F. Pires[3]

Universidade Federal da Paraíba

carlos@ccen.ufpb.br
Universidade Federal do Rio Grande do Norte

eduardoafs@ppgsc.ufrn.br, {thais,jair}@ufrnet.br
Universidade Federal do Rio de Janeiro

{paulo.f.pires, fdelicato}@gmail.com

Abstract. Este artigo apresenta UbiAcme, uma linguagem de descrição arquitetural para sistemas ubíquos que foi construída baseada em trabalhos que estabelecem características de sistemas ubíquos. UbiAcme estende a linguagem Acme adicionando abstrações para: (i) representar contexto como abstração de primeira ordem no nível arquitetural, (ii) expressar parâmetros de qualidade, como Qualidade de Serviço (QoS) e Qualidade de contexto (QoC) e (iii) estabelecer situações de reconfiguração dinâmica. Para avaliar a linguagem proposta, é usada uma aplicação ubíqua, o *Smart Car*.

Keywords. Arquitetura de Software, Computação Ubiqua, Linguagem de Descrição Arquitetural

1 Introdução

A Computação Ubíqua [1] utiliza uma grande variedade de dispositivos, sensores e redes que interagem para formar um ambiente distribuído e heterogêneo, integrado às atividades diárias dos usuários. Tipicamente, aplicações ubíquas são compostas por *serviços*, fornecidos por diversos provedores; são *sensíveis ao contexto*, ou seja, usam informações de contexto para a realização das suas tarefas e para se adaptar a mudanças no seu ambiente de execução; são móveis, de forma a oferecer os serviços em qualquer lugar que o usuário precise.

Para atender a essas características, as aplicações ubíquas precisam possuir capacidade de adaptação sem perder qualidade de serviço. Ou seja, elas precisam usar informações do contexto de forma a modificar-se dinamicamente, adaptando à ambientes heterogêneos e a novos provedores e mantendo a qualidade dos serviços, em conformidade com os requisitos de cada aplicação. Uma informação de contexto é qualquer informação que pode ser usada para caracterizar a situação de uma entidade, que pode ser uma pessoa, lugar, objeto ou o próprio sistema, considerada relevante para a interação entre um usuário e a aplicação [2]. Dentre essas informações, destacam-se os parametros de qualidade, que são atributos responsáveis por medições de qualidade acerca dos serviços utilizados e providos pelas aplicações [3].

No cenário das aplicações ubíquas, é essencial representar essas informações que dão suporte a ubiquidade desde as fases iniciais do projeto de desenvolvimento de software. Tais elementos incluem informações de contexto, especificações de adaptações previsíveis, parâmetros de qualidade e outras.

Lopes e Fiadeiro [15] discutem sobre a importância da representação do contexto como elemento de primeira ordem na arquitetura, sob a ótica de que a arquitetura do sistema depende do contexto no qual ele se encontra. Para isso, são necessários mecanismos para representação de contexto que permitam o suporte a dinamicidade no nível arquitetural,

essencial em sistemas cientes ao contexto. Uma vez que sistemas ubíquos enquadram-se nessa classificação de sistemas, é fundamental representar dados de contexto a nível arquitetural.

No contexto de arquitetura de software as linguagens de descrição arquitetural (ADL – *Architecture Description Language*) [4, 5, 6] são linguagens de alto nível para representar e analisar projetos arquiteturais, provendo um framework conceitual e uma notação sintática para caracterizar arquiteturas de software em termos de seus elementos e relacionamentos. Essas linguagens essencialmente oferecem abstrações para representação da arquitetura de um software através de componentes, conectores e configurações. Os *componentes* representam as funcionalidades do software, os *conectores* são elementos de comunicação entre componentes, e a *configuração* é utilizada para descrever o relacionamento entre componentes e conectores. Além de representar os componentes essenciais, uma ADL pode oferecer recursos para a especificações de informações adicionais que possam ser utilizadas durante o projeto arquitetural ou em situações de adaptação quando a arquitetura precisa ser modificada. No caso de aplicações ubíquas, no entanto, a literatura não reporta ADLs específicas que disponibilizem abstrações para representar informações essenciais para tais aplicações, como metadados e representação de contexto.

Este trabalho tem como objetivo preencher essa lacuna, propondo UbiAcme, uma ADL para modelar aplicações ubíquas que consiste em uma extensão da ADL Acme [7], incluindo elementos importantes para descrição desse tipo de aplicações. A linguagem Acme foi escolhida por ser uma ADL de propósito geral que fornece um quadro estrutural simples para representar arquiteturas, e permitir a integração de diferentes ferramentas oferecendo uma forma comum de intercâmbio de informações arquiteturais. Nesse sentido, UbiAcme propõe adicionar elementos para: (i) representar *contexto* como abstração de primeira ordem no nível arquitetural, (ii) expressar parâmetros de qualidade, como QoS e QoC e (iii) definir situações de reconfiguração dinâmica em função dos contextos e dos parâmetros de qualidade. Para avaliar a linguagem proposta, será usada uma aplicação ubíqua, a *Smart Ca*r [8, 9, 10].

Este artigo está estruturado da seguinte forma. A Seção 2 apresenta os conceitos básicos relacionados com computação ubíqua e linguagem de descrição arquitetural. A Seção 3 introduz UbiAcme, descrevendo seus elementos e construtores. Na Seção 4 é apresentada a aplicação usada para ilustrar o uso da linguagem: o *Smart Car*. A Seção 5 contém alguns trabalhos relacionados. A Seção 6 contém as conclusões e direções para trabalhos futuros.

2 Fundamentação sobre Computação Ubíqua

Esta seção apresenta os conceitos básicos relacionados à computação ubiqua, que são fundamentais para a compreensão deste trabalho.

Há muitos desafios para viabilizar a computação ubíqua [11]: (i) lidar com diversos tipos de eventos, tais como mudança de ambiente, eventos de intercâmbio de dados (como envio de sinais) e eventos específicos de domínio, (ii) adaptar o sistema em tempo de execução, (iii) integrar vários tipos de elementos computacionais, tais como telefones inteligentes, sensores, atuadores e (iv) gerenciar a comunicação entre os elementos, permitindo mobilidade e segurança. As soluções para esses desafios em geral não são triviais e tipicamente envolvem vários elementos em cooperação para apoiar a operação do sistema.

O desenvolvimento de sistemas ubíquos envolve vários domínios da computação, tais como inteligência artificial, engenharia de software, seguranças de redes e sistemas distribuídos. Nesse cenário multi-disciplinar, faz-se necessário sistematizar a construção das partes constituintes de um sistema ubíquo, definindo, no nivel arquitetural, os elementos do sistema, como também as interações entre eles. A definição de documentos de arquitetura de software pode, além de satisfazer essa necessidade de sistematização, permitir avaliações do sistema antes da sua implementação.

2.1 Características de Sistemas Ubíquos

Em Machado et al [12] foi conduzida uma revisão sistemática que identificou um conjunto de 10 (dez) elementos comuns aos sistemas ubíquos em geral. Tais elementos foram

comparados com as características essenciais estabelecidas no trabalho de Spínola e Travassos [13], ilustrados na Tabela 1, e identificou-se que os elementos relacionados atendem as características essenciais de sistemas ubíquos.

Características	**Descrição**
Onipresença de Serviço	O sistema deve permitir ao usuários mover-se com a sensação de levar os serviços de computação com ele.
Invisibilidade	O sistema deve evitar, do ponto de vista do usuário, a sensação de uso explícito de um computador
Sensibilidade ao contexto	O sistema de ser capaz de coletar informações do ambiente em que ele está sendo usado.
Comportamento adaptativo	O sistema deve ser capaz de, dinamicamente, adaptar os serviços oferecidos de acordo com o ambiente em que ele está sendo usado, respeitando suas limitações
Captura de Experiência	O sistema deve ser capaz de capturar e registrar experiências para uso posterior.
Descoberta de serviços	O sistema deve descobrir serviços de acordo com o ambiente em que ele está sendo usado.
Composição	O sistema deve possuir a capacidade de compor serviços, com base em serviços básicos, para criar um serviço solicitado pelo usuário.
Interoperabilidade espontânea	O sistema deve ter a capacidade de mudar suas funcionalidades durante o seu funcionamento de acordo com o ambiente em que se encontra.
Heterogeneidade de dispositivos	O sistema deve ser capaz de utilizar diversos dispositivos e ajustar-se a cada um deles.
Tolerância a falhas	O sistema deve ser capaz de recuperar-se quando ocorre falhas, tais como problemas de conectividade com os serviços

Tabela 1. Características de projetos de sistemas ubíquos

As características listadas pela Tabela 1 foram utilizadas como base para este trabalho, no sentido de fornecer uma fundamentação acerca de quais recursos deveriam ser providos por ADLs, conforme será discutido na Seção 3.

2.2 Identificação de Elementos a serem providos pela ADL

Em termos de representação no nível arquitetural, a ADL visa expressar a arquitetura da aplicação em si, que envolve representar e utilizar informações relacionadas a contextos. Nesse sentido, discutiremos a seguir quais as características listadas na Tabela 1 que são necessárias, no nível arquitetural, para descrever aplicações ubíquas. Outros aspectos descrito na Tabela 1, como, por exemplo Onipresença de Serviços, Heterogeneidade de Dispositivos, Invisibilidade, entre outros, estão relacionados com a infraestrutura de suporte a computação ubíqua usada pelas aplicações.

A característica de **Onipresença de Serviço** está diretamente associada a plataforma de implementação usada pelo sistema, pois depende de como os serviços estão instalados nos ambientes e da interface que esses serviços proveem. Portanto, não é representada na descrição arquitetural da aplicação. Da mesma forma, **Invisibilidade** também não é representada, por estar diretamente associada aos elementos físicos e plataforma de execução que compõem o sistema, uma vez que essa caracteristica está relacionada a forma como é feita a interação com o usuário – podendo ser explítica (através do uso de teclado, mouse ou painéis) ou implícita (através de sensores). **Sensibilidade ao Contexto** deve ser representada na arquitetura como elemento de primeira ordem, uma vez que o contexto influencia o comportamento do sistema, conforme discutido em [15] e, portanto, é necessário a sua representação explícita. **Comportamento Adaptativo** também deve ser representado no nível arquitetural, uma vez que a especificação deve usar mecanismos de adaptação arquitetural para dar suporte a adaptação em tempo de execução. A representação de contextos e associação explícita do efeito da mudança de contexto sobre a arquitetura do sistema é a forma mais natural de representar essa característica em nível de arquitetura de software. **Captura de Experiência** pode ser representada, entretanto não será foco no presente trabalho por envolver conceitos relacionados com Inteligência Artificial. A **Descoberta de Serviço** deve ser representada no nível arquitetural, pois envolve mecanismos de conexões dinâmicas que relacionam contextos (que devem ser representados como elementos de primeira ordem) a mecanismos de comunicação em si. Não existem, entretanto,

abordagens consolidadas para representação de descoberta de serviço no nível arquitetural. **Composição de Função** é uma característica que pode ser representada em ADLs através de composição de componentes. **Heterogeneidade de Dispositivos** é naturalmente representada em ADLs, uma vez que os elementos de processamento, serviços, sensores, atuadores e todos os demais elementos relacionados a computação ubíqua são representados como componentes e conectores. Por fim, **Tolerância a Falhas** pode ser representada em uma ADL como um conceito transversal, entretanto se encontra fora do escopo deste trabalho.

3 UbiAcme

Linguagens de Descrição Arquitetural (*Architecture Description Languages* – ADLs) [4, 5, 6] são usadas para especificação de documentos de arquitetura de software e permitem a descrição de sistemas através de representação de componentes, conectores e configuração, responsável por estabelecer como os componentes comunicam-se através dos conectores. Dentre as ADLs existentes, Acme [7] destaca-se por ser uma linguagem genérica, com uma extensão que permite definição de restrições a partir de expressões em lógica de primeira ordem [14], permitindo também a definição de restrições no nível de estilo (ou família) arquitetural. Acme fornece uma ontologia com oito conceitos utilizados em uma representação arquitetural. (i) *Components* (componentes) representam os componentes arquiteturais, que são abstrações computacionais responsáveis por processamento, armazenamento ou manipulação de dados. Componentes podem ter múltiplas interfaces, chamadas port (portas); (ii) *Connector* (conectores) é a entidade responsável por representar a interação entre componentes. Conectores estabelecem uma relação de comunicação direta entre dois ou mais componentes. Conectores possuem um conjunto de interfaces, chamadas roles (papéis); (iii) *Systems* são abstrações para definição de sistemas, que por sua vez são constituídos de um conjunto de componentes, um conjunto de conectores e um conjunto de attachments que descrevem a topologia do sistema; (iv) *Ports* (Portas) são as interfaces dos componentes, conforme supracitado, cada porta identifica um ponto de interação entre o componente e seu ambiente. Um componente pode prover múltiplas interfaces usando diferentes tipos de portas; (v) *Roles* (Papéis) são as interfaces dos conectores; (vi) *Representation* (Representação) são elementos que permitem a descrição de visões arquiteturais, essa visões podem: (a) ser usadas exibir a mesma arquitetura sob uma perspectiva diferente, ou (b) para detalhamento, descrevendo a organização interna de um elemento arquitetural (componente, conector, porta, ou papel); (vii) *Attachments* definem um conjunto de associações entre as portas dos componentes e os papéis dos conectores; (viii) *Properties* (propriedades) acomodam uma ampla variedade de informações auxiliares, em geral anotações sobre elementos ou informações detalhadas, geralmente não estruturais. Embora os elementos originalmente descritos na ontologia de Acme sejam suficientes para definir a estrutura de uma arquitetura de software, a natureza das características arquiteturais de aplicações ubíquas vai além do que é atualmente suportado nessa ADL e em outras da literatura. Nesse contexto, faltam modelos que capturem totalmente informações específicas da Computação Ubíqua, tais como: a representação das informações de contexto e a qualidade das informações de contexto relacionadas aos serviços sensíveis ao contexto usados pela aplicação.

3.1 Elementos de UbiAcme

A linguagem UbiAcme foi definida de modo a possibilitar a descrição de aplicações ubíquas considerando as características supra relacionadas. Dentre as abstrações apresentadas por UbiAcme, destacam-se os elementos para: (i) representação de contexto, (ii) representação de meta-dados – dados sobre serviços e qualidade de informação de contexto, (iii) reconfiguração dinâmica. Esta seção discute as alterações realizadas por UbiAcme dividindo-as em quatro tópicos principais: **Contexto**, **Parametros de Qualidade**, ***Attachment* Condicional** e **Reconfiguração Dinâmica**. A Tabela 2 lista todas as novas abstrações introduzidas na linguagem, que serão discutidos ao longo desta seção, apresentando um breve resumo de sua funcionalidade.

Elementos	Função
Context	Representar contextos no nível arquitetural
ContextExpression	Descrever conjunto de condições que caracterizam as circunstâncias que ativam um contexto (Context)
OnActivate	Descrever um conjunto de adaptações arquiteturais que serão realizadas no instante em que o Context for ativado
OnDeactivate	Descrever um conjunto de aptações arquiteturais que serão realizadas no instante em que o Context for desativado. Esse elemento não é obrigatório no corpo de um Context
Undo	Desfazer todas as adaptações realizadas por um *OnActivate*, com exceção das adaptações persistentes.
Persistent	Indica que a adaptação arquitetural realizada ao ativar de um Context persistirá mesmo após a desativação desse Context.
QoSParameter	Representar parâmetros de qualidade de serviço que estão associados a um provedor de serviço (componente ou conector).
QoCParameter	Representar parâmetros de qualidade de contexto que estão associados a uma interface (port ou role).
Attachment Condicional	Attachment que só está ativo – i.e. permite o fluxo de informações – em determinadas situações que são definidas a partir dos contextos ativos e inativos.
On .. do ..	Especificar uma condição de adaptação para o sistema e descreve as alterações (adaptações) arquiteturais que serão realizadas quando a condição de adaptação acontecer.
detach	Destroi conexões existentes entre componentes do sistema
remove	Destroi componentes, conectores ou propriedades do sistema

Tabela 2. Elementos adicionados a UbiAcme e um resumo de sua funcionalidade.

Contexto.

Contextos são representados em UbiAcme através do elemento ***Context*** o qual possui essencialmente uma expressão de ativação de contexto, que é uma expressão boleana envolvendo propriedades, parâmetros de qualidade e estados de outros contextos (ativo ou inativo) que, quando satisfeita, representa a ativação de um contexto, i.e. o sistema encontra-se em um conjunto de circunstâncias pré-definidas. Assume-se que todo e qualquer contexto está desativado quando sua expressão de ativação não é satisfeita. O elemento *Context*, que representa contextos no nível arquitetural, possui um ***ContextExpression,*** responsável por descrever um conjunto de condições através de expressões booleadas que caracterizam as condições que ativam um *Context*, utilizando propriedades, parâmetros de qualidade ou até o estado (ativo/inativo) de outros *Contexts*. A Fig. 1. mostra a definição sintática para o elemento *Context*, definido essencialmente a partir de uma *ContextExpression*. Contextos também podem possuir propriedades (como qualquer elemento Acme) e permitem definições de *OnActivate* e *OnDeactivate*, que serão explicados a seguir.

```
<ContextDefinition>::=
  Context ID = "{"
    ContextExpression = <ContextExpressionDef> ";"
    [<OnActivateDef> ";"]
    [<OnDeactivateDef> ";"]
    (<PropertyDefinition>)*
  "}" [";"]
<OnActivateDef>::= OnActivate "{"
    ([Persistent] <ElementDef>
    | <PlastikExpression> )*
  "}"
<OnDeactivate> ::= OnDeactivate "{"
    [Undo ";"]
    (<PlastikExpression>)*
  "}"
```

Fig. 1. Definição sintática para o element *Context*

O elemento ***OnActivate*** descreve um conjunto de adaptações arquiteturais que serão realizadas no momento em que o *Context* for ativado. Essas adaptações incluem definição de novos elementos, criação ou eliminação de conexões e alterações no comportamento de elementos existentes. Por outro lado, o elemento ***OnDeactivate*** descreve um conjunto de adaptações arquiteturais que serão realizadas no instante em que o *Context* for desativado. Esse elemento não é obrigatório no corpo de um *Context*, e por padrão todas as adaptações realizadas por um *OnActivate* são desfeitas no momento em que o *Context* é desativado. Entretanto, na definição de um *OnDeactivate* é possível incluir o comando ***Undo***, que desfaz todas as alterações realizadas no *OnActivate* do contexto em questão.

Durante a definição do elemento *OnActivate* é possível utilizar a palavra-chave ***Persistent*** para discriminar as adaptações que irão persistir mesmo após a desativação do *Context*. Elementos *Persistent* são uma alternativa mais simples para o *OnDeactivate* em situações nas quais o arquiteto não quer que todas as adaptações arquiteturais se percam ao desativar do *Context*.

A Fig. 2 mostra um exemplo de definição de *Contexto* para a aplicação *Smart Car*, onde é definido o contexto *Highway_Entering*, que representa a situação na qual o *Smart Car* encontra-se entrando em uma auto-estrada, o que significa que os módulos de piloto automático não estão prontos e precisam ser carregados. Esse contexto estará ativo quando o contexto *Highway_Driving* não estiver ativo e o sistema de navegação indicar que o veículo está em uma auto-estrada. O contexto *Highway_Driving* ativo, por sua vez, representa a circunstância na qual o *Smart Car* está em modo de piloto automático, ativado em auto-estradas. Quando o contexto *Highway_Entering* for ativado, será definido um componente persistente, *DrivingSystem*, que possui uma propriedade booleana *isReady* e três portas: *wheelController*, responsável por enviar sinais de controle de volante; *speedController*, responsável por enviar sinais de controle de velocidade; e *drivingEvents*, responsável por receber eventos e dados de elementos externos. *DrivingSystem* foi definido como um componente persistente pois implementa um complexo sub-sistema de Inteligência Artificial e, por isso, demanda um tempo maior para ser totalmente inicializado. Esse componente é responsável pelos controles automáticos do carro (o piloto automático). Implementando-o como componente persistente, descreve-se explicitamente que esse componente só será inicializado quando o contexto *Highway_Driving* tornar-se ativo. Entretanto, após a inialização, *DrivingSystem* irá permanecer no sistema, auxiliando o usuário na navegação e controles do veículo e possibilitando a ativação do piloto automático mais rápido em próximas entradas em auto-estradas. Uma alternativa seria inicializar o *DrivingSystem* durante as atividades normais do carro, entretanto essa solução iria alocar recursos computacionais desnecessariamente, na maioria das situações, remover o *DrivingSystem*, por outro lado, iria requerer uma nova inicialização sempre que o contexto *Highway_Entering* tornar-se ativo.

```
17  Context Highway_Entering = {
18    ContextExpression not(Highway_Driving) AND NavigationSystem.locationId=HIGHWAY;
19    OnActivate {
20        Persistent Component DrivingSystem = {
23            Property isReady : boolean;
24            Port wheelController;
25            Port speedController;
26            Port drivingEvents;     }
27        Attachment DrivingSystem.drivingEvents to EventConnector.eventProvider;
28        EventMonitor.synchronizeNav = true; }}
```

Fig. 2. Exemplo de elemento *Context*

Parâmetros de Qualidade.

UbiAcme define duas abstrações para descrição de parametros de qualidade: *QoSParameter* e *QoCParameter*. O elemento ***QoSParameter*** permite estabelecer quais são os medatados de qualidade de serviços das informações de contexto de um serviço. O elemento ***QoCParameter*** permite que, para um dado parâmetro de contexto essencial a aplicação, estabeleçam-se os critérios de qualidade relacionados a ele. A Fig. 3 exibe a definição sintática para os elementos *QoSParameter* e *QoCParameter*, que consistem apenas

da palavra-chave *QoSParameter* ou *QoCParameter* seguido de um nome identificador. É possível definir um tipo e o valor padrão para cada parâmetro. Qualquer tipo de *Property* pode ser atribuído aos construtores *QoSParameter* e *QoCParameter*, esse tipo será responsável por definir o tipo de valor que esse parametro armazena. Quando atribuído, o valor padrão de um *QoSParameter* ou *QoCParameter* deve estar de acordo com o tipo definido.

```
<QoSParameterDef>::=
   QoSParameter ID [":" <PropertyTypeID>] ["=" <value>]
";"
<QoCParameterDef>::=
   QoCParameter ID [':" <PropertyTypeID>] ["=" <value>]
";"
```

Fig. 3. Definição Sintática dos elementos *QoSParameter* e *QoCParameter*

Ambos, *QoSParameter* e *QoCParameter,* estendem a abstração *Property*, adicionando as características de dinamicidade e monitoramento contínuo, além de explicitar que os valores armazenados em *QoSParameters* e *QoCParameters* serão calculados pela aplicação, em tempo de execução. Quando não for possível calcular o valor do parâmetro, o valor padrão é atribuído. Elementos *QoSParameter* podem ser instanciados em elementos arquiteturais que podem ser usados para representar provedores de serviço, que típicamente são elementos *Component* ou *Connector*. Elementos *QoCParameter*, por sua vez, só podem ser instanciados em interfaces (*Port* ou *Role*), devido a sua natureza de estarem diretamente associados aos dados de serviços, que são necessariamente providos através de interfaces. A Fig. 4 mostra um exemplo de utilização de um *QoSParameter* e *QoCParameter*.

```
1 Component Type LocationService = {
2    QoSParamenter availability;
3    Port provideLocation = {
4          QoCParamenter precision;   }
5    Property syncInterval = 30;
6 }
```

Fig. 4. Exemplo de uso de *QoSParameter* e *QoCParameter*

Como exemplo, na Fig. 4, define-se o *Component Type LocationService*, que é o tipo básico dos componentes provedores de serviço de localização. Esse tipo de componente possui: (i) um *QoSParameter availability*, responsável por monitorar a disponibilidade do servidor; (ii) uma interface para o serviço *provideLocation*, essa interface (*Port*) possui um *QoCParameter precision*, que se refere a precisão do dado que é fornecido pelo serviço; e (iii) uma propriedade *syncInterval*, que define o intervalo de atualização dos parametros de qualidade.

Attachment Condicional.

Outro elemento definido por UbiAcme é o ***Attachment* Condicional**. Esse elemento tem comportamento semelhante ao *Attachment* nativo de Acme, entretanto só está ativo – i.e. permite o fluxo de informações – em situações que são definidas a partir de uma expressão de contexto, a qual consiste em uma combinação de estados (ativo ou inativo) de contextos pré-definidos. A Fig. 5 exemplifica um *Attachment* condicional, definindo três *attachments* que se encontram ativos apenas quando em contextos específicos. Na Fig. 5, os contextos *UsingGPSLocation*, *UsingGSMLocation* e *UsingWifiLocation* são ativados a depender principalmente do parêmetro de QoC (*QoCParamenter*) *precision*. Da forma como foram definidos, no máximo um desses contextos pode estar ativo por vez. A partir do estado desses contextos (ie. ativo ou inativo), são realizados *attachments* condicionais que relacionam o provedor do serviço de localização (que pode ser GPS, GSM ou Wifi) ao conector *Location*, que conecta o serviço de localização aos elementos que utilizam esse serviço.

```
01  Context UsingGPSLocation = {
02    ContextExpression LocationGPS.error != 0  AND
03          LocationGPS.provideLocation.precision >= LocationGSM.provideLocation.precision AND
04          LocationGPS.provideLocation.precision >= LocationWifi.provideLocation.precision;
05   }

06  Context UsingGSMLocation = {
07    ContextExpression LocationGSM.error !=0 AND
08          LocationGSM.provideLocation.precision > LocationGPS.provideLocation.precision AND
09          LocationGSM.provideLocation.precision >= LocationWifi.provideLocation.precision;
10  }

11  Context UsingWifiLocation = {
12     ContextExpression LocationWifi.error != 0  AND
13           LocationWifi.provideLocation.precision > LocationGPS.provideLocation.precision AND
14           LocationWifi.provideLocation.precision > LocationGSM.provideLocation.precision;
15  }

16  Attachment LocationGPS.provideLocation to Location.provider  [UsingGPSLocation];
17  Attachment LocationGSM.provideLocation to Location.provider  [UsingGSMLocation];
18  Attachment LocationWifi.provideLocation   to Location.provider  [UsingWifiLocation];
```

Fig. 5. Exemplo de *Attachment* condicional

Reconfiguração Dinâmica.
Devido a natureza dinâmica de aplicações ubiquas, faz-se necessária a inclusão de mecanismos arquiteturais de planejamento de reconfiguração dinâmica programada. Originalmente, Acme/Armani não oferecem recursos para se descrever o planejamento de reconfigurações programadas, por isso foram adicionadas a UbiAcme as extensões propostas por Plastik [16]. A primeira extensão é um comando condicional ***On-Do*** que permite ao arquiteto expressar em qual situação a reconfiguração programada deve ocorrer e o que deve ser mudado para efetivar essa reconfiguração. A segunda extensão é um par de construtores que permite a alteração da arquitetura: (i) ***Detach***, utilizado para remover uma ligação entre uma port e uma role; (ii) ***Remove***, usado para destruir um componente, conector, representação, propriedade ou *QoSParameter* e *QoCParamenter*.

```
01  On (not(UsingGPSLocation) and not(UsingGSMLocation) and not(UsingWifiLocation)) do {
02          Component DefaultLocationProvider : LocationService;
03          Attachment DefaultLocationProvider.providerLocation to Location.provider;
04  }
```

Fig. 6. Exemplo da utilização do comando *On-do*

Na Fig. 6, o comando condicional *On-Do* (linha 1) verifica se os três contextos relacionados ao serviço de localização não estão ativos. Se essa condição for verdadeira, significa que nenhum serviço de localização está sendo utilizado, então deve ser realizada uma reconfiguração que consiste em incluir um provedor de localização padrão e associar seu serviço de localização a *Role Provider* do *Connector Location*.

A BNF completa da ADL UbiAcme está disponível no endereço:
`http://consiste.dimap.ufrn.br/projects/ubiacme/`

4 Aplicação Ubíqua em UbiAcme: *Smart Car*

De modo a verificar a expressividade da linguagem para especificar um sistema ubíquo, usamos a aplicação de *Smart Car*, baseada em propostas existentes para um veículo autonômico. No setor automotivo, os carros autônomos são dotados de sistemas embarcados que caracterizam um sistema computacional com propósitos específicos [8]. O carro pode enviar sinais a outros automóveis automaticamente para indicar as condições da estrada à frente ou a necessidade de frear, por exemplo.

A arquitetura do sistema *Smart Car*, nesse estudo de caso, possui 7 componentes: (i) sub-sistema de condução, responsável pelo piloto automático; (ii) sub-sistema de navegação, responsável por controlar os elementos físicos do carro, como por exemplo alterando sua

velocidade e direção, esse sub-sistema monitora dados do veículo, do clima e da pista que serão exibidos para o usuário e utilizados pelo sistema na tomada de decisão; (iii) mecanismo de pilotagem, responsável por captar as instruções do usuário, através de sensores no volante e pedais; (iv) componente de comunicação via rádio, responsável pela interação com outros sistemas semelhantes; (v) sub-sistema de localização, responsável por monitorar a localização do veículo; (vi) serviços de localização, responsáveis por prover a localização, esses serviços são implementados nos próprios elementos de hardware do sistema; e (vii) módulo de eventos, responsável por tratar requisições assincronas, disparadas segundo alguma condição pré-estabelecida. Para essa aplicação foram definidos inicialmente dois cenários principais, usados como fundamento para a descrição arquitetural. Esses cenários definem situações nas quais o *Smart Car* poderá estar inserido (contexto), como também descreve ações que devem ser tomadas pelo sistema computacional do veículo. No primeiro cenário, o *Smart Car* deve ser capaz de ativar o piloto automático quando o veículo entra em uma auto-estrada. É importante ressaltar que algumas propostas definem o Smart Car como um carro totalmente automático, [17], entretanto, optou-se por descrever uma proposta mais realista, que é um veículo semi-autônomo. O segundo cenário descreve o uso e sincronização do Serviço de Localização utilizado pelo veículo, que se mostrou a situação mais comum nesse tipo de aplicação ciente de contexto.

4.1 Cenário 1 – Piloto automático em auto-estradas

Definição: O veículo entra em uma auto-estrada, o sistema do SmartCar recupera todas as informações da estrada, como: tráfego (número de carros por metro quadrado), velocidade máxima, condições climáticas (temperatura, umidade, etc), condições da pista (acesso, visibilidade, etc). Enquanto isso, o sub-sistema de condução, que irá assumir o controle da velocidade e direção do carro, é preparado e sincronizado com o sub-sistema de navegação, responsável por controlar os elementos físicos do carro e monitorar elementos de navegação, como posicionamento e condições dos equipamentos. Quando ambas as etapas estão concluídas, o sistema desabilita as conexões do mecanismo de pilotagem, responsável por captar os controles do usuário através da direção e pedais, ativando assim o piloto automático.

No piloto automático, o sistema usa baixas frequências de rádio para se comunicar com os carros próximos.

Descrição: Para esse cenário, o sistema reage a dois contextos: *Highway_Entering*, que inicializa os módulos do piloto automático e é ativado quando o carro entra em uma auto-estrada e o componente responsável pelo piloto automático não está pronto para uso; e *Highway_Driving*, que ao ser ativado inicializa o monitor de eventos com a comunicação pelo rádio e ativa os módulos do piloto automático para controlar toda navegação do carro. *Highway_Driving* é ativado quando o carro está em uma auto-estrada e o componente responsável pelo piloto automático já está inicializado. Enquanto o contexto *Highway_Driving* estiver ativo, o carro está sendo controlado automaticamente.

A Fig. 7 mostra a descrição arquitetural, em UbiAcme, do conjunto de elementos arquiteturais responsáveis pelo tratamento do cenário 1. Nessa descrição, existem 3 componentes fundamentais, que são os componentes que representam os elementos físicos dos pedais e do volante (*Wheel* e *Pedals*), e o componente que representa o sub-sistema de navegação (*NagivationSystem*), que atua sobre o motor, freios e rodas do veículo para realizar as operações de mudança de direção ou velocidade. O conector *PilotingMechanism* é responsável pela comunicação entre os controladores do veículo e o atuador *NavigationSystem*, para que o veículo responda de acordo com os comandos. Na Fig. 7, os contextos *Highway_Entering* e *Highway_Driving* são descritos. O contexto *Highway_Entering* define um componente persistente, *DrivingSystem*, que será conectado ao *PilotingMechanism* e assumirá o controle do carro durante o contexto *Highway_Driving*. O contexto *Highway_Driving*, por sua vez, define um componente *radioCommunicator*, que será utilizado para comunicação com outros sistemas semelhantes que se encontram próximos.

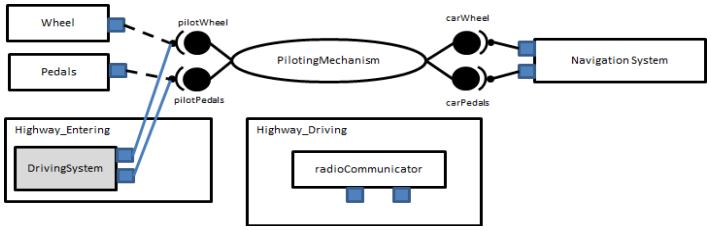

Fig. 7. Representação gráfica da descrição do cenário 1

```
1  System Smart_Car {
2     Component NavigationSystem = {
3          Property locationId;
4          Port wheelControl;
5          Port speedControl; }
6     Component Wheel = { Port wheel; }
7     Component Pedals = { Port pedals; }
8     Connector PilotingMechanism = {
9          Role pilotWheel;
10         Role carWheel;
11         Role pilotPedals;
12         Role carPedals; }
13    Attachment PilotingMechanism.carWheel to NavigationSystem.wheelControl;
14    Attachment PilotingMechanism.carPedals to NavigationSystem.speedControl;

15    Attachment Wheel.wheel to PilotingMechanism.pilotWheel [not(Highway_Driving)];
16    Attachment Pedals.pedals to PilotingMechanism.pilotPedals [not(Highway_Driving)];
```

Fig. 7a. Definição dos elementos principais da arquitetura responsável por implementar o cenário 1

```
17  Context Highway_Entering = {
18     ContextExpression not(Highway_Driving) AND NavigationSystem.locationId=HIGHWAY;
19     OnActivate {
20        Persistent Component DrivingSystem = {
23           Property isReady : boolean;
24           Port wheelController;
25           Port speedController;
26           Port drivingEvents;     }
27        Attachment DrivingSystem.drivingEvents to EventConnector.eventProvider;
28        EventMonitor.synchronizeNav = true; }}
```

Fig. 7b. Descrição do contexto Highway_Entering

```
28  Context Highway_Driving = {
29     ContextExpression DrivingSystem.isReady AND EventMonitor.syncReady AND
                  NavigationSystem.locationId=HIGHWAY;
30     OnActivate {
31        Attachment DrivingSystem.wheelController to PilotingMechanism.pilotWheel;
32        Attachment DrivingSystem.SpeedCotroler to PilotingMechanism.pilotPedals;
33        Component radioCommunicator = {
34           Port radioEvent;
35           Port radioData; }
36         Attachment radioCommunicator.radioEvent to   EventConnector.eventProvider; }
37     OnDeactivate {
38        Undo;
39        Remove DrivingSystem;
40        Remove AutoPilot; } } }
```

Fig. 7c. Descrição do contexto Highway_Driving

4.2 Cenário 2 – Serviço de Localização

Definição: Em intervalos fixos (30 segundos), o sub-sistema de localização do veículo sincroniza todos os meta-dados relativos aos serviços de localização, que podem ser por GPS, GMS ou Wifi. A partir desses dados, o sistema identifica o melhor serviço de localização baseado em sua precisão, adaptando-se para utilizar apenas esse serviço. Se nenhum dos serviços estiver disponível, o sistema realiza uma adaptação para inserir um

componente que é capaz de fornecer um dado de localização estimado, podendo basear-se nas posições prévias e velocidade do veículo.

Descrição: Para esse cenário foram definidos 6 contextos: (i) *ChoosingLocationProvider*, que será ativado a cada 30 segundos para atualizar os meta-dados dos serviços de localização (que serão providos pelos próprios serviços) e para escolher o serviço a ser utilizado; (ii) *UsingGPSLocation*, que é ativado quando o sistema escolhe utilizar o mecanismo de localização via GPS; (iii) *UsingGSMLocation*, que é ativado quando o sistema escolhe utilizar o mecanismo de localização via GSM; (iv) *UsingWifiLocation*, ativado quando o sistema escolhe utilizar o mecanismo de localização via Wifi; (v) *NoLocationProvider*, ativado quando o sistema identifica que nenhum dos servidores de localização está disponível; e finalmente, (vi) *LocationServiceError*, ativado sempre que o serviço atual de localização entra em um estado de erro.

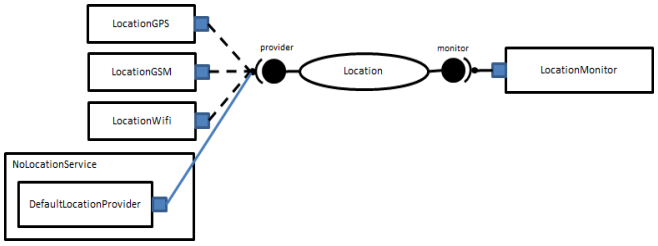

Fig.8. Representação gráfica do cenário 2

```
1  System Smart_car
2  Component Type LocationService = {
3     QoSParamenter availability;
4     Port provideLocation = {
5           QoCParamenter precision;   }
6     Property syncInterval = 30;
7  }
8  Component LocationMonitor = {
9     Port locationPort;
10     Property error : integer;
11 }
12 Connector Location = {
13    Role provider;
14    Role monitor;
15 }
16 Component LocationGPS : LocationService = { ... }
17 Component LocationGSM : LocationService  = { ... }
18 Component LocationWifi : LocationService  = { ... }
19 Attachment LocationMonitor.locationPort to Location.monitor;
20 Attachment LocationGPS.provideLocation to Location.provider
                                        [ChoosingLocationProvider OR UsingGPSLocation];
21 Attachment LocationGSM.provideLocation to Location.provider
                                        [ChoosingLocationProvider OR UsingGSMLocation];
22 Attachment LocationWifi.provideLocation to Location.provider
                                        [ChoosingLocationProvider OR  UsingWifiLocation];
```

Fig. 8a. Descrição dos principais elementos arquiteturais responsáveis por implementar o cenário 2

```
23  Context ChoosingLocationProvider = {
24     ContextExpression syncInterval == 0;
25     OnActivate {  }
26  }
```

Fig. 8b. Descrição do contexto *ChoosingLocationProvider*

```
27  Context UsingGPSLocation = {
28     ContextExpression LocationGPS.error != 0  AND
29              LocationGPS.provideLocation.precision >= LocationGSM.provideLocation.precision AND
30              LocationGPS.provideLocation.precision >= LocationWifi.provideLocation.precision;
31  }
```

Fig. 8c. Descrição do contexto *UsingGPSLocation*

```
32 Context UsingGSMLocation = {
33    ContextExpression LocationGSM.error !=0 AND
34              LocationGSM.provideLocation.precision > LocationGPS.provideLocation.precision AND
35              LocationGSM.provideLocation.precision >= LocationWifi.provideLocation.precision;
36 }
```

Fig.8d. Descrição do contexto *UsingGSMLocation*

```
37 Context UsingWifiLocation = {
38    ContextExpression LocationWifi.error != 0  AND
39              LocationWifi.provideLocation.precision > LocationGPS.provideLocation.precision AND
40              LocationWifi.provideLocation.precision > LocationGSM.provideLocation.precision;
41 }
```

Fig.8e. Descrição do contexto *UsingWifiLocation*

```
42 Context NoLocationService = {
43      ContextExpression not(ChoosingLocationProvider) AND not(UsingGPSLocation)
                                AND not(UsingGSMLocation)  AND not(UsingWifiLocation);
44      OnActivate {
45      Component DefaultLocationProvider : LocationService;
46      Attachment DefaultLocationProvider.providerLocation to Location.provider; }
47 }
```

Fig.8f. Descrição do contexto *NoLocationService*

```
48 Context LocationServiceError = {
49   ContextExpression ((UsingGPSLocation AND LocationGPS.error == 0) OR
                        (UsingGSMLocation AND LocationGSM.error == 0) OR
                        (UsingWifiLocation AND LocationWifi.error==0)) AND not(ChoosingLocationProvider);
50   OnActivate {
51      Property syncInterval=0;  }
52 }
```

Fig.8h. Descrição do contexto *LocationServiceError*

Essencialmente, a arquitetura possui um componente *LocationMonitor*, responsável por monitorar os dados de localização e repassar as informações para as demais partes do sistema; e um conector *Location*, que realiza a comunicação entre o serviço de localização atual e o *LocationMonitor*. A Fig. 8 mostra uma representação gráfica para essa descrição e os trechos da descrição UbiAcme relativos à esse cenário. Nessa figura é possível observar os componentes *LocationGPS*, *LocationGSM* e *LocationWifi*, responsáveis por prover o serviço de localização, que estão conectados ao conector *Location* por um *attachment* condicional, que é ativo quando o serviço respectivo é escolhido para ser utilizado pelo sistema. No contexto *NoLocationService* é instanciado um novo componente, que é conectado ao conector *Location*, esse componente é o *DefaultLocationProvider*, que é responsável por implementar um mecanismo padrão de localização.

5 Trabalhos Relacionados

No contexto de ADLs para computação ubíqua, existem poucas propostas, sendo elas ScudADL [18] e LindaQoS [19]. Nessa seção, ambas serão discutidas e será feita uma breve relação entre seus elementos e os elementos de UbiAcme.

ScudADL [18] é uma ADL para descrição de middlewares adaptativos para computação ubiqua, baseando-se em π-ADL [20] e D-ADL [21] para representar ambos estrutura e comportamento dos elementos arquiteturais que compõem o sistema. ScudADL possui ênfase em reconfiguração dinâmica, e é voltada para descrição de ambientes inteligentes. A linguagem não possui abstrações específicas para computação ubíqua, não inclui elementos para representação de contexto, representação de parametros de qualidade e conexões condicionais (*attachments* condicionais). Em comparação com UbiAcme, a linguagem mostra-se bastante complexa, por se basear em π-calculus. Por outro lado, ScudADL é capaz de representar em detalhes a estrutura e o comportamento dos elementos do sistema,

enquanto UbiAcme possui ênfase em descrições estruturais. Uma vez que ScudADL não possui abstrações para representação de contexto - pois a linguagem é baseada em expressões de reconfiguração -, torna-se inviável reutilizar ou enriquecer contextos.

LindaQoS (LINguagem de Descrição de Arquitetura com QoS) [19] propõe uma linguagem específica de domínio voltada para representação de parâmetros de qualidade e reconfiguração dinâmica. A linguagem dá margem a representação de contexto de forma implícita, e permite a definição de estruturas de reconfiguração baseada nessas definições. Uma vez que a ênfase da linguagem é representar parâmetros de QoS, a definição de contexto, em LindaQoS, mostra-se menos robusta que em UbiAcme, que possui um elemento arquitetural de primeiro nível para representar contextos. Outro ponto negativo é a dificuldade em reusar e rastrear expressões de contextos, uma vez que essas expressões podem se encontrar espalhadas ao longo da descrição.

Ambos os trabalhos não satisfazem todas as características desejáveis de ADLs para computação ubíqua, que são: (i) representações para contexto como elemento de primeira ordem; (ii) mecanismo para adaptações em tempo de execução; (iii) mecanismo para reconfiguração arquitetural em tempo de execução; e (iv) representação de parâmetros de qualidade no nível arquitetural. LindaQoS implementa as características (ii), (iii) e (iv), enquanto ScudADL implementa as características (ii) e (iii). A caracteristica (i), cuja importância é discutida em [15], não é provida por nenhuma das ADLs avaliadas.

6 Conclusões e Trabalhos Futuros

Este artigo propõe uma extensão para a ADL Acme com recursos para representar na descrição arquitetura elementos comuns de aplicações ubíquas. Acme foi escolhida por ser uma ADL de propósito geral que fornece um quadro estrutural simples para representar arquiteturas, possibilitando a integração de diferentes ferramentas através uma forma comum de intercâmbio arquitetural.

A partir de um estudo prévio [12], identificou-se os elementos arquiteturais que aparecem na maior parte dos sistemas ubiquos existentes, e que, portanto, oferecem soluções comuns para os problemas relacionados a esse domínio. O conhecimento estrutural e comportamental desses elementos, aliado às características de projetos de sistemas ubíquos identificados por [13], possibilitou identificar características desejáveis em ADLs para descrição arquitetural desse tipo de sistema. Definiu-se então a ADL UbiAcme que estende a ADL Acme, implementando o suporte às características identificadas como desejáveis em ADLs para computação ubíqua. Como elementos notáveis de UbiAcme destacam-se: (i) a representação de contexto como metadado arquitetural de primeira ordem, que permite reuso de definições desses contextos, além da possibilidade de descrever explicitamente a interação dos elementos da arquitetura com os contextos e si.; (ii) a inserção de elementos para representar parâmetros de qualidade, que são atributos dinâmicos monitorados em tempo de execução que podem ser utilizado em conjunto com os contextos para disparar adaptações arquitetuais; (iii) suporte a reconfiguração dinâmica a partir de expressões de contextos e propriedades, que possibilita ao sistema uma grande dinamicidade em tempo de execução. Esses elementos são usados em conjunto com os elementos de Acme, dotando a linguagem de meios para tratar as restrições e as características de aplicações ubíquas.

Como principal trabalho futuro, está em desenvolvimento a ferramenta UbiAcme Studio, que será capaz de criar, editar e visualizar de maneira textual e/ou gráfica, descrições arquiteturais em UbiAcme. Outro importante trabalho em andamento é a validação da linguagem que será feita através de análises experimentais controladas utilizando cenários reais.

Agradecimentos

Este trabalho recebeu financiamento parcial do CNPq Proc. Num. 560266/2010-3 de Thais Batista, e da Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) projetos de Flávia C. Delicato e Paulo F. Pires.

Referências

1. Weiser, M. The computer for the 21st century. SIGMOBILE Mob. Comput. Commun. Rev., ACM, 1999, 3, 3-1.
2. Dey, A.; et al.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Journal of Human-Computer Interaction 16(2), pp.97-166, 2001.
3. Batista, C. et al. Monitoramento de metadados para computação ubíqua. In: XXXIX Seminário Integrado de Software e Hardware (SEMISH 2012), Curitiba, PR, Brasil.
4. Shaw, M. and Garlan, D.: Software Architecture: Perspectives on an emerging discipline. Prentice-Hall, Upper Saddle River, NJ, USA. 1996.
5. Medvidovic, N. and Taylor, R. N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26, 1, 70–93. 2000.
6. Clements, P.: A survey of architecture description languages. In Proceedings of the 8th International Workshop on Specification and Design (IWSSD'96). IEEE Computer Society, Washington, DC, USA, 16–25. 1996.
7. Garlan, D., Monroe, R. and Wile, D.: Acme: An Architecture Description Interchange Language. In: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research. IBM Press, 1997.
8. Horn, P.: Autonomic computing: IBM's Perspective on the State of Information Technology. 2001.
9. Kephart, J. e Chess, D.: The vision of autonomic computing. Computer, 36(1):41–50. 2003.
10. Jung, C.; Osório, F. S.; Kelber, C.; Heinen, F. Computação embarcada: Projeto e implementação de veículos autônomos inteligentes.
11. Kumar, S.: Challenges for Ubiquitous Computing, Proceedings of the Fifth International Conference on Network and Services (ICNS'09), 2009, pp. 526-535.
12. Machado C. at al, "Architectural Elements of Ubiquitous systems. A Systematic Review", Proceedings of The Eighth International Conference on Software Engineering Advances (ICSEA'13), 2013.
13. Spínola, R. and Travassos, G.: "Towards a framework to characterize ubiquitous software projects", Information and Software Technology, v. 54, 2012, pp. 759-785.
14. Monroe, R.: Capturing software architecture expertise with Armani. Technical report, School of Computer Science, Carnegie Mellon University, USA, 1998.
15. Lopes, A. and Fiadeiro, J. L.: Context-awareness in software architectures. In: MORRISON, R.; OQUENDO, F. (Ed.). EWSA. [S.l.]: Springer, 2005. (Lecture Notes in Computer Science, v. 3527), p. 146_161. ISBN 3-540-26275-X.
16. Batista, T., Joolia, A. and Coulsen, G.: Managing Dynamic Reconfiguration in Component-based Systems, Computer Science Department , Federal University of Rio Grande do Norte (UFRN), 59072-970, Natal - RN, Brazil
17. Karvonen, H.; Kujala, T. and Saariluoma, P.: In-Car Ubiquitous Computing: Driver Tutoring Messages Presented on a Head-Up Display Intelligent Transportation Systems Conference, 2006. ITSC '06. IEEE, 2006, 560-565
18. Wu, Q. and Li, Y.: ScudADL: An architecture description language for adaptive middleware in ubiquitous computing environments. Computing, Communication, Control, and Management, 2009. CCCM 2009. ISECS International Colloquium on, 2009, 4, 611-614
19. Neto, C.; Rodrigues, R. and Soares, L. F. G.: Architectural description of QoS provisioning for multimedia application support. Multimedia Modelling Conference, 2004. Proceedings. 10th International, 2004, 161-166
20. Oquendo, F.: Pi-ADL: an Architecture Description Language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures SIGSOFT Softw. Eng. Notes, 2004, 1-14
21. Li Changyun, Li Gansheng, He Pinjie: Formal Dynamic Architecture Description Language D-ADL, Journal of Sotware, VOL.17, NO.6, pp. 1349-1359, 2006.