Interpretação de Contexto no Ambiente SmartAndroid

Matheus Erthal[1], **Orlando Loques**[1]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)
Niterói – RJ – Brazil

{merthal,loques}@ic.uff.br

Resumo. *Aplicações cientes de contexto vêm adquirindo cada vez mais visibilidade no atual cenário de desenvolvimento de aplicações. Visando a construção de aplicações cientes de contexto no ambiente de uma smarthome, o SmartAndroid propõe uma plataforma para a interação entre diferentes dispositivos oferecendo abstrações de alto nível para os programadores e interfaces de prototipagem para usuários domésticos. Este trabalho detêm um maior foco na declaração e exposição da informação de contexto, e na interpretação da mesma através de regras, que podem ser criadas tanto pela aplicação quanto através de uma interface gráfica amigável a usuários não especializados.*

1. Introdução

A Computação Ubíqua, como proposta por Weiser na década de 90 [Weiser 1991], prevê uma mudança no paradigma de interação entre o usuário e os sistemas computacionais. Ao invés de se interagir com computadores diretamente, através da Computação Ubíqua usuários interagem com computadores de forma indireta. O sistema identifica as necessidades do usuário obtendo informação de contexto através de sensores, e provê serviços através de atuadores.

Uma subárea da Computação Ubíqua de particular interesse nesse trabalho é chamada de Computação Ciente de Contexto [Dey et al. 2001], na qual sistemas computacionais adaptam automaticamente seu comportamento de acordo com circunstâncias físicas. Tais circunstâncias podem, em princípio, ser qualquer coisa fisicamente mensurável ou detectável, como a presença da pessoa, a hora do dia ou condições atmosféricas [Coulouris et al. 2005].

Com os recentes avanços na tecnologia de computação móvel, assim como na comunicação sem fio de dispositivos, abriu-se espaço para o crescimento da Computação Ubíqua [Coulouris et al. 2005, Lyytinen and Yoo 2002], que pode também ser chamada de Computação Pervasiva [Saha and Mukherjee 2003, Satyanarayanan 2001], Inteligência Ambiente (AmI) [Augusto and McCullagh 2007], ou outros [Ranganathan et al. 2005, Augusto and McCullagh 2007]

O SmartAndroid visa providenciar um suporte conceitual e de implementação para a construção de aplicações cientes de contexto no ambiente de *smarthome*. Para que a informação de contexto seja facilmente manipulada pelas aplicações e pelos usuários do sistema, este trabalho procura prover uma maneira fácil de se expor as informações e as funcionalidades dos recursos do ambiente, além de prover uma abstração que eleve o contexto para um nível mais alto.

2. SmartAndroid

O SmartAndroid é uma plataforma que provê facilidades e padrões de programação a aplicações cientes de contexto focadas no ambiente de *smarthomes*.

Ao invés de se lidar diretamente com sensores e atuadores espalhados no ambiente, o SmartAndroid utiliza, como abstração, componentes chamados Agentes de Recurso (AR). Os ARs são elementos de primeira ordem da plataforma, representando recursos de hardware ou software disponíveis no ambiente. Eles têm como responsabilidade coletar as diversas medidas dos sensores ou componentes de *software* presentes no contexto de execução do das aplicações [Bezerra 2011]. Os ARs resolvem a complexidade de tratamento com os componente de baixo nível e disponibilizam interfaces padrões de interação tanto local como em rede.

No ambiente de smarthome, podem ser criados ARs para representar, por exemplo: uma televisão que expõe suas informações de contexto e provê serviços para as aplicações; sensores de temperatura da casa; sensores de vazamento de gás na cozinha; um medidor de pressão arterial; um dispositivo para fechar as janelas; etc.

Para possibilitar a comunicação entre os diferentes ARs, o SmartAndroid dispõe de um serviço que gerencia estes recursos, e mantém informações de identificação e localização dos mesmos, dentre outras. Este serviço chama-se Serviço de Gerenciamento de Agentes de Recurso (SGAR) e é instanciado apenas uma vez no sistema distribuído. Como apresentado na Figura 1, o SGAR é composto por três componentes, ou serviços: o Serviço de Registro de Recurso (SRR), o Serviço de Descoberta de Recursos (SDR) e o Serviço de Localização de Recursos (SLR). A diferença entre o SDR e o SLR está no fato de que o primeiro se refere à localização na rede de computadores, e o segundo se refere à localização física do dispositivo no ambiente.

O SGAR tem por objetivo aumentar a visibilidade das informações de contexto dos ARs e tornar a comunicação transparente para o programador das aplicações.

Na Figura 1, aparecem no item (a) os serviços do SGAR; no item (b) tem-se o ambiente funcionando em outro terminal; e os itens (c) e (d) apenas representam a existência de diversos outros ARs.

2.1. Comunicação

O SmartAndroid provê dois tipos comunicação: síncrona e assíncrona. A comunicação síncrona corresponde ao acesso direto entre ARs. A comunicação assíncrona ocorre através da notificação por eventos.

Através da chamada síncrona pode-se fazer também comunicação em grupo. Primeiramente deve-se conhecer o AR chamado (chamada de um para um), ou tipo do AR (chamada de um para muitos), ou referenciar a todos os ARs (chamada de um para todos), esta informação pode ser adquirida através do SDR.

A comunicação assíncrona corresponde ao padrão *publish-subscribe* (publica-subscreve), onde um AR interessado ("*stakeholder*") subscreve a outro, e quando a informação está pronta para ser disponibilizada, notifica os subscritores.

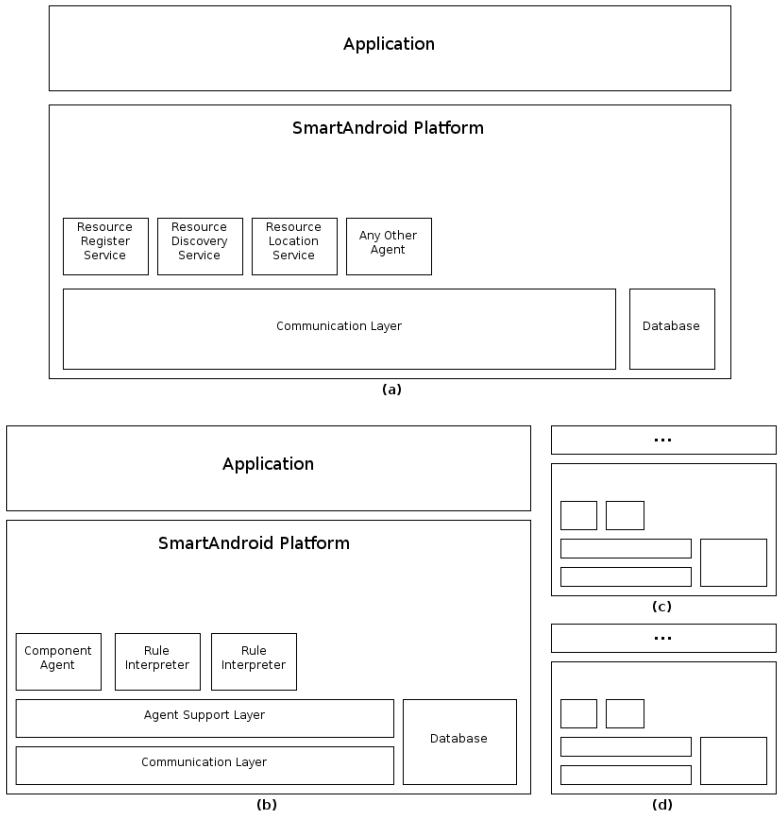

Figura 1. Arquitetura em Camadas do SmartAndroid

3. Contexto

O contexto exerce um papel de fundamental importância na Computação Ubíqua. Durante a década de 90, diversos autores procuraram definir contexto, muitas delas pouco apuradas. Baseado em definições de autores anteriores, Dey e Abowd [Dey et al. 2001] foram capazes de compor a seguinte definição para contexto:

> "Qualquer informação que pode ser usada para caracterizar a situação de entidades (i.e., pessoa, lugar ou objeto) que são consideradas relevantes para a interação entre um usuário e uma aplicação, incluindo o próprio usuário e aplicação. Contexto é tipicamente a localização, identidade, e estado de pessoas, grupos, e objetos computacionais e físicos"

Segundo [Dey et al. 2001] três entidades de contexto foram identificadas como de maior importância, são elas: lugares, pessoas e coisas. "Lugares" são regiões geográficas, como salas, escritórios, prédios e ruas. "Pessoas" podem ser tanto indivíduos como grupos, co-localizados ou distribuídos. "Coisas" são objetos físicos ou componentes de *software*.

Após a identificação das entidades, são introduzidas quatro categorias de contexto essenciais: identidade, localização, estado e tempo. "Identidade" corresponde à identificação única de uma entidade. "Localização" diz respeito não só a posição de uma entidade, mas orientação, proximidade em relação a outra entidade, etc. "Estado" identifica características intrínsecas da entidade (e.g., temperatura de uma sala, sinal vital

de uma pessoa). “Tempo” é uma informação de contexto quando ajuda a caracterizar a situação, geralmente usado com outras informações de contexto, indicando um instante ou período durante o qual alguma informação contextual é conhecida ou relevante.

3.1. Variáveis de Contexto

No SmartAndroid, as entidades cujo contexto é acessado, são modeladas em ARs, e suas informações de contexto são as Variáveis de Contexto (VC) expostas pelos mesmos. Acessando as VCs de um AR obtém-se alguma informação em específico de estado do mesmo. Se, por exemplo, há um agente para a televisão registrada no sistema, possivelmente ele provê VCs para qual a programação que está sendo exibida, qual a programação agendada, se a própria televisão está ligada, se está gravando alguma programação, etc.

As VCs possibilitam que as informações de contexto sejam acessadas sem que haja um componente centralizado armazenando-as e disponibilizando-as, o que poderia incorrer em um *overhead* na comunicação.

Os VCs são as portas de saída do AR, são declarados programaticamente no momento da criação do mesmo, e podem ser acessados em tempo de execução.

Na Subseção 4.1 fala-se mais sobre as VCs, explicando como é feita a declaração das mesmas, uso e a implementação destas no sistema.

3.2. Serviços

Enquanto as VCs possibilitam o acesso às informações de contexto de um AR, e têm uma semântica de leitura associada, os Serviços de um AR são suas interfaces de comando, ou seja, suas funcionalidades, e possuem uma semântica de escrita.

O papel dos serviços é permitir que a aplicação interaja ativamente com o ambiente, mudando assim o próprio contexto. Por exemplo, uma televisão integrada ao sistema pode oferecer serviços como para desligá-la, mudar de canal, gravar alguma programação, mostrar uma mensagem na tela, perguntar algo ao usuário, gerar um alerta, pausar a programação, etc.

Os Serviços são as portas de entrada de um AR, e, como as VCs, são declarados programaticamente em tempo de compilação, e podem ser acessados em tempo de execução.

A Figura 2 apresenta um AR com suas portas de saída e entrada. Podem ser vistas os Serviços como portas de entrada e as VCs como portas de saída. Pode ser vista ainda a porta de saída para registro do AR e as portas usadas na comunicação assíncrona. O AR dispõe para comunicação assíncrona uma porta de entrada para receber subscrições (“*Subscription*”), e uma de saída para notificar (“*Notify*”); uma porta de saída para se subscrever a outro AR (“*Subscribes*”), e uma de entrada para receber as noticações (“*Notify*”).

3.3. Interpretação

A interpretação tem a função de aumentar a abstração em torno das VCs. Geralmente aplicações necessitam de informações em um nível mais alto do que sensores proveem. Como um sensor é representado no sistema através do AR que o controla, uma outra

Figura 2. Portas de um Agente de Recurso

abstração é usada para representar a informação interpretada, chamado de Interpretador de Regra (IR).

A interpretação de contexto possibilita que diversas informações de contexto, relacionadas através de uma expressão lógica, sejam manipuladas em uma entidade em separado, provendo separação de interesses. O interessado poderá referenciar apenas uma entidade, ao invés de várias, a fim de se obter a informação interpretada.

Como foi representado na Figura 3, o IR se posiciona entre os ARs e a aplicação, aumentando assim o nível de abstração.

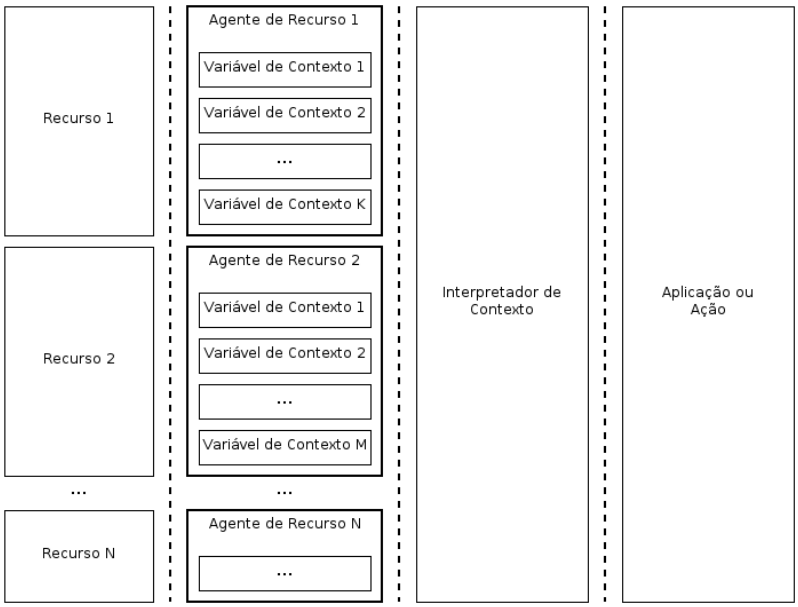

Figura 3. Camada de Interpretação de Regra

3.3.1. Criação de Interpretadores de Regras

Para se implementar a interpretação das VCs, devem ser criados agentes IRs. Fazem parte da obrigação de um interpretador: ler e compilar uma expressão de regra definida, monitorar as VCs de interesse, avaliar a expressão e notificar os interessados na regra recém interpretada.

A expressão do RI é armazenada em uma estrutura de dados no RI, e possui uma sintaxe em específico, que relaciona as VCs através de operadores de comparação e lógicos. Uma VC pode ser comparada com: um valor constante, uma outra VC, uma expressão isolada por parênteses; ou pode ainda ter seu valor avaliado como um valor booleano. Definimos como operadores de comparação os seguintes: $=, \neq, <, >, \leq$ e $\geq$. Como operadores lógicos, definimos os seguintes: $\wedge$ ("E", "AND"), $\vee$ ("OU", "OR") e $\sim$ ("NÃO", "NOT").

Considere a seguinte regra: "determinada pessoa está dormindo e a pressão arterial estiver alta, ou determinada pessoa está acordada e pressão arterial está baixa". Para modelar esta regra podem ser utilizadas as VCs "dormindo" do agente "Pessoa" (e utilizar o operador $\sim$ nesta VC para saber se a pessoa está acordada), e a VC "pressão arterial" do agente do medidor de pressão, onde "alta" e "baixa" devem ser definidas como uma constante. Assim, podemos reescrever a regra como:
("dormindo" $\wedge$ "pressão arterial" > 150.0) $\vee$ ($\sim$"dormindo" $\wedge$ "pressão arterial" < 70.0).

A criação da regra do IR pode ser feita duas maneiras: em nível mais baixo, ao se definir a expressão rigorosamente obedecendo à sintaxe, ou através de uma interface gráfica. Os usuário do SmartAndroid, mesmo sem qualquer especialização, irão dispor de uma interface gráfica para facilitar a criação de regras de contexto em tempo de execução. Através da interface gráfica o usuário poderá em alto nível escolher VCs de interesse e relacioná-las utilizando os operadores de comparação, e relacionar estas subexpressões com operadores lógicos.

Não só os recursos do ambiente são modelados como AR, mas também o IR, assim sendo, se utiliza dos mecanismos de comunicação de maneira homogênea em relação aos outros ARs. Isto possibilita que a aplicação consulte ou se subscreva ao IR da mesma maneira que faria com os outros ARs, e permite também que o IR se subscreva aos outros ARs.

Quando criado, o IR se registra junto ao Serviço de Registro de Recursos, como qualquer outro AR, podendo, em seguida, ser encontrado através do Serviço de Descoberta de Recursos.

3.3.2. Interpretador de Regra em Funcionamento

Estes agentes são responsáveis pela coleta das diferentes VCs, potencialmente distribuídas, e de avaliar a regra que relaciona estas VCs, notificando em seguida aos seus subscritores, ou *stakeholders*.

Uma regra que acabou de ser executada volta a executar dado um evento de modificação de uma das VCs, ou seja, se a regra é notificada de uma atualização de uma VC, ela avalia sua expressão de novo. Uma regra também pode ser executada de novo

caso a aplicação requisite.

A Figura 4 representa o roteiro de funcionamento do IR em etapas, são elas:

1. Notificação por parte das VCs dos ARs de interesse
2. Diferentes notificações chegam pela mesma porta, assim, nesta etapa é feita o tratamento da notificação
3. O módulo de avaliação está ligado à regra armazenada, e, no momento de chegada de uma nova informação de contexto, avalia a regra novamente
4. Se a regra for validada, passa-se ao processo de notificação dos interessados
5. O grupo de interessados é notificado

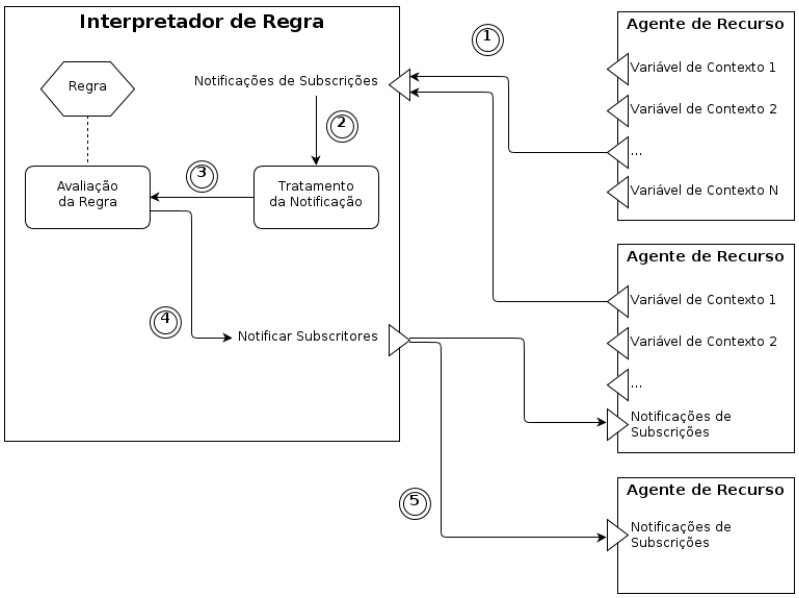

Figura 4. Interpretador de Regra

Observa-se que, conforme o IR é também um AR, isto cria um padrão descentralizado de processamento de regras, não havendo, portanto, um motor de regras centralizado. Esta arquitetura permite que as regras possam ser executadas em qualquer dispositivo, não havendo necessidade de um dispositivo mais robusto para processar todas, se houverem muitas. Também é possível que um novo dispositivo sendo introduzido no sistema já contenha regras próprias.

Os benefícios da utilização de IRs são vários. Dentre eles podemos citar a separação de interesses. Não é necessário ao programador da aplicação se preocupar em subscrever a todas as VCs de interesse e implementar a lógica da regra, basta definir uma expressão para a regra. Além disso, o programador não tem de se preocupar em implementar funções para alteração da regra em tempo de execução, para desativar a regra, etc, pois isso já é provido pelo ambiente.

3.4. Temporização

A componente Tempo na interpretação, ajuda a caracterizar o contexto. Por exemplo, saber se usuário está dormindo, é uma informação de contexto, mas também é uma informação de contexto saber se o usuário está dormindo a 1 hora inteira, e pode ser que só a segunda seja de interesse à aplicação.

A Figura 5 inclui o componente temporizador na, já apresentada, Figura 4. Durante a etapa de avaliação da regra, se esta contém temporizadores na declaração, são feitos agendamentos no temporizador, a fim de se contar a passagem de tempo e continuar avaliando a regra.

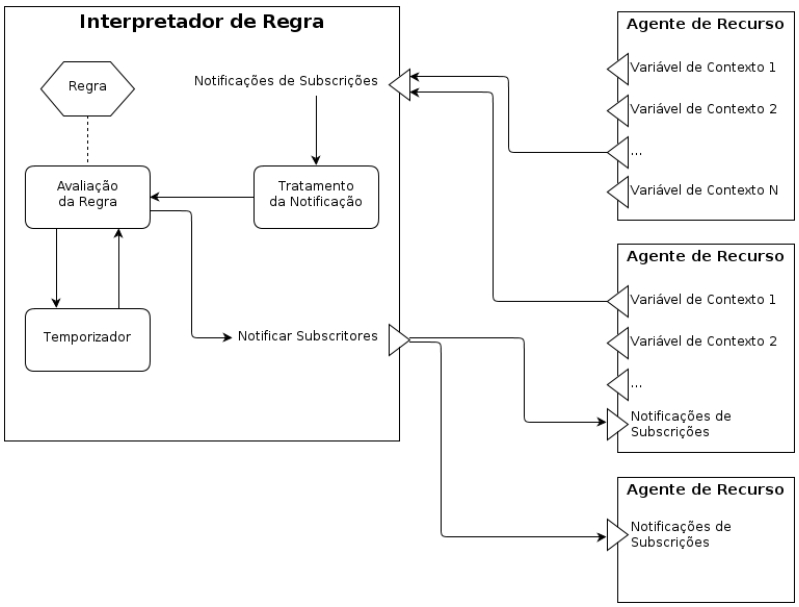

Figura 5. Interpretador de Regra com temporizador

O restante do procedimento de interpretação é equivalente à interpretação de regras não temporizadas, como explicitado na Subseção 3.3.2.

A sintaxe da regra deve ser capaz de permitir uma fácil declaração do temporizador e associação à expressão. Assunto este que será tratado na subseção de implementação dos temporizadores, a Subseção 4.4.

4. Implementação

A API do SmartAndroid, como é evidente, está sendo construída para funcionar na plataforma Android, utilizando seus padrões de implementação, bibliotecas do *framework* e design de interfaces, e a linguagem Java foi adotada para a programação. Além dos simuladores estão sendo utilizados um conjunto de quatro celulares (com Android 2.3) e três tablets (com Android 3.2).

4.1. Anotações de Variáveis de Contexto e Serviços

A criação de um AR é feita estendendo-se a classe abstrata *ResourceAgent*. Esta classe implementa alguns métodos para a interação com outros ARs e características pertinentes a todos os ARs, como nome, endereço, tipo, id, etc. Uma aplicação que deseja implementar um AR para a interação com algum dispositivo em específico, deve implementar uma classe que estenda *ResourceAgent*. Como exemplo, temos a classe "Stove" apresentada na Figura 6. Esta classe é implementada pelo criador da aplicação de SmartHome e usuário da API SmartAndroid. Para se criar um novo AR para "Stove", basta instanciar esta classe, e registrá-la junto ao SRR.

```
public class Stove extends ResourceAgent implements IStove {
    private float tempCooktop1 = 0.0f;
    ...
    private float tempOven1 = 0.0f;
    private float gasLeak = 0.0f;
    private boolean onCooktop1 = false;
    ...
    private boolean onOven1 = false;

    @ContextVariable(name = "Temperatura boca 1", type = CVType.Temperature)
    public float getTemperatureCooktop1() {
        ...
        return this.tempCooktop1;
    }

    @ContextVariable(name="Temperatura forno", type = CVType.Temperature)
    public float getTemperatureOven() {
        ...
        return this.tempOven1;
    }

    @ContextVariable(name="Está Ligado", description="", type = CVType.On)
    public boolean getIsOn() {
        ...
        return this.isOn;
    }

    ...
```

Figura 6. Implementação do Agente de Recurso

A Figura 6 mostra que são declarados atributos correspondentes as propriedades dos "fogões", sendo elas: a temperatura das bocas ("*tempCooktop1*"), a temperatura do forno ("*tempOven1*"), o sensor de vazamento de gás natural ("*gasLeak*"), o sensor de bocas ligadas ("*onCooktop1*") e o sensor de forno ligado ("*onOven1*").

O acesso à informação de contexto do AR é feito através de chamadas de métodos remotos, e o AR se encarrega de acessar o sensor e ler as informações.

Os métodos do AR que possibilitam a leitura de seu estado interno, ou seja, de sua informação de contexto, são as VCs deste AR. Para declarar um método como VC, basta associar a anotação "*ContextVariable*" ao método, como mostrado na Figura 6. Observa-se a anotação sobre os métodos "*getTemperatureCooktop1*", "*getTemperature-Oven*" e "*getIsOn*".

O AR, na anotação de VC, deve definir um nome único, para que não haja duas diferentes VCs com o mesmo nome em um AR. Pode-se declarar opcionalmente uma descrição em alto nível da VC e o tipo da mesma.

A declaração de um Serviço, como descrito na Subseção 3.2, é feita de maneira idêntica à declaração de uma VC. Um método é criado para implementar o serviço, e uma anotação do tipo "Service" é agregado a este método, passando o nome e a descrição do serviço.

A Figura 7 mostra como foram implementadas as anotações na interface IResourceAgent, implementada por todos os ARs. Foram incluídas na interface tanto a anotação para VC quanto para Serviço, permitindo que qualquer AR os implemente.

O uso de anotações permite facilmente se distinguir os métodos que devem ser expostos no AR dos outros métodos internos do AR, sem haver confusão. Assim, os métodos internos do AR que não devem ser acessados por chamadas de métodos remotos, não o serão. O uso de anotações possibilita ainda incluir informações adicionais, no caso:

```java
public interface IResourceAgent {
    public String getResourceClassName();
    public enum CVType { Null, On, OFF, Temperature, Light, Open, Close, Gas, Wet };
    public enum SType { Null, Video, Audio, ShortText, LongText, Alarm, Beep, TurnOn, TurnOff }

    @Retention(RetentionPolicy.RUNTIME) // Make this annotation accessible at runtime via reflection.
    @Target({ ElementType.METHOD }) // This annotation can only be applied to class methods.
    public @interface ContextVariable {
        String name();
        String description() default "";
        CVType type() default CVType.Null;
    }

    @Retention(RetentionPolicy.RUNTIME) // Make this annotation accessible at runtime via reflection.
    @Target({ ElementType.METHOD }) // This annotation can only be applied to class methods.
    public @interface Service {
        String name();
        String description() default "";
        SType type() default SType.Null;

    }

    public void registerStakeholder(String method, String url);
    public void notificationHandler(String change);
}
```

Figura 7. Implementação das Anotações

o nome da VC ou Serviço, descrição e tipo.

A biblioteca “Java Annotations”, assim como as classes “Method” e “Class” fazem parte da biblioteca Java Reflection. Com uma instância de um Method em específico e um objeto da classe que implementa este método, é possível invocar este método. Assim, para se consultar uma VC basta um *Method* (com a anotação *ContextVariable*) e um objeto do AR. O AR pode ser obtido com a identificação do mesmo e um acesso ao Serviço de Descoberta de Recurso, logo, a Variável de Contexto é compreendida pelo par Method e ID do AR.

4.2. Inicialização do Interpretador de Regra

A interpretação de VCs, como descrita na Subseção 3.3, se dá através da criação de interpretadores. Esta criação, por sua vez, ocorre por meio da instanciação de um objeto da classe *RuleInterpreter*, o que possibilita a criação de interpretadores em tempo de execução.

O diagrama de sequência da Figura 8 apresenta o processo de criação de um IR, a partir da aplicação (“*Application*”). Este processo envolve a definição de uma expressão no IR, que é analisada (“*parse(expr)*”) e armazenada em uma estrutura de dados interna. As VCs referenciadas na expressão da regra são uma a uma subscritas pelo IR onde, primeiramente, os ARs detentores das VCs são procurados, através de “*search(rai)*”, e em seguida o IR se subscreve na VC em específico através de “*subscribe(this, cv)*”. O início da operação ocorre quando o IR é inicializado (“*start*”), fazendo com que o agente do mesmo se registre no sistema.

O diagrama em árvore da Figura 9 mostra a estrutura de dados que armazenará a expressão exemplo da Subseção 3.3.1, após compilada. Uma árvore é gerada para que possa ser facilmente avaliada, a partir das folhas para a raiz.

Na árvore são identificados dois tipos de nós: nós de comparação e nós de relação lógica. Nos nós de comparação implementa-se igualdades e desigualdades entre objetos comparáveis, ou seja, uma VC e uma constante, ou uma VC e outra. Caso a VC seja booleana, pode-se só avaliar a mesma, sem que seja feita uma comparação com um valor booleano.

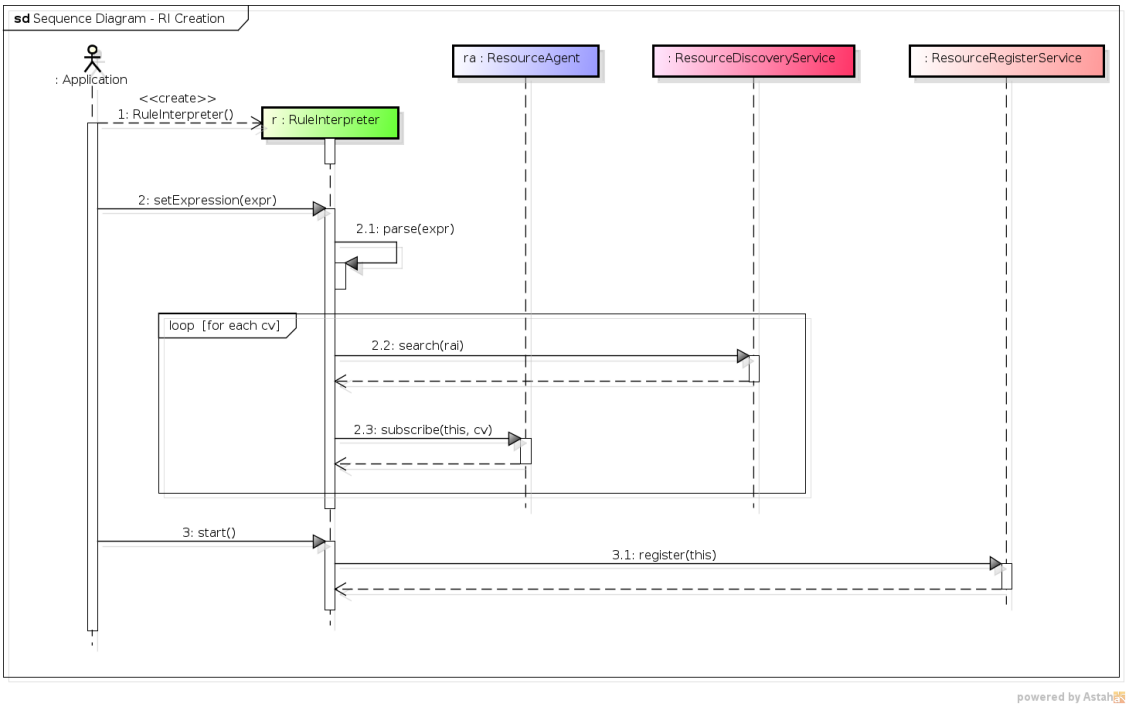

Figura 8. Criação de um Interpretador de Regra

4.3. Interpretador de Regra em Funcionamento

Os IRs criados são passivos, e só atuam em decorrência de uma notificação de uma VC subscrita ou quando o limite do temporizador foi alcançado, portanto, estes agentes não detêm uma *thread* só para si, economizando recursos do sistema. O IR pode executar em qualquer terminal que esteja estabelecido. Se estiver co-localizado em relação às suas VCs, então diminuirá também o fluxo de mensagens na rede – se uma chamada é feita para o mesmo terminal, ela é convertida em uma chamada local, não utilizando a infraestrutura de rede.

A execução do IR está representada no diagrama de sequência da Figura 10. A execução pode ser separada em três etapas: a ligação com o IR, a avaliação da regra pelo IR e a notificação da validade da regra. A primeira é idêntica para qualquer AR, a busca no Serviço de Descoberta de Recursos e subscrição à VC exposta pelo IR, mas como o IR possui apenas uma VC não foi necessário esta identificação.

A segunda etapa compreende a avaliação da regra. Na Figura 10 esta etapa se inicia com a notificação do IR por AR que sofreu uma atualização em uma VC de interesse do IR. Se não há temporizadores na expressão da regra, esta é apenas avaliada e, se retornou verdadeiro, passa-se a etapa três, de notificação aos interessados. Se houver temporizadores na expressão, então inicia-se um ciclo de avaliação onde o IR passa faz agendamentos e reavaliações da regra. A temporização será melhor explicada na Subseção 4.4.

4.4. Temporização

A fim de se aumentar o poder de expressividade das expressões de regras, pode-se associar um temporizador a cada VC, a cada subexpressão, ou mesmo à expressão completa.

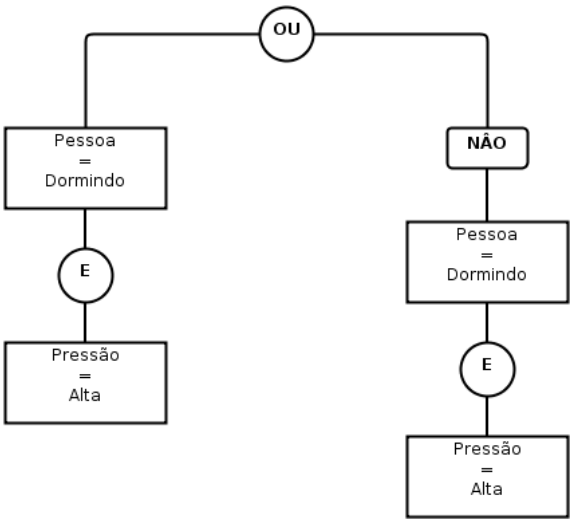

Figura 9. Diagrama em árvore do exemplo de regra

O significado de se associar um temporizador à uma VC implica dizer que o valor desta VC não pode mudar do início da temporização até o fim da contagem. Por exemplo, desejamos monitorar se o usuário está dormindo à 1 hora. Neste caso a VC de interesse é "dormindo", de pessoa, e o valor é "verdadeiro". Se o valor mudar, o temporizador pode ser reinicializado.

Se o temporizador estiver associado à expressão completa, então a avaliação deve garantir que a expressão se mantenha verdadeira no decorrer do tempo, independentemente da mudança nos valores das VCs. Ou seja, ainda que uma VC mude seu valor, não necessariamente o temporizador precisa ser reinicializado. Uma avaliação será feita para garantir que a expressão continua verdadeira.

Caso o temporizador esteja associado à uma parte da expressão, ou subexpressão, a técnica funciona de maneira semelhante à temporização da expressão completa.

Se a temporização foi interrompida por motivo de atualização nas variáveis de contexto então, dado o momento da validação da regra como verdadeira, o temporizador é re-inicializado.

Pode acontecer de mais de um temporizador estar associado a partes da regra, e definidos com tempos diferentes. Neste caso, o temporizador de maior tempo começa a contar primeiro, e em seguida, no seu devido tempo, os outros temporizadores vão sendo inicializados. Ao final da contagem, todos os temporizadores terminarão ao mesmo tempo, a regra é completamente avaliada e, se validada, os subscritores são notificados. Embora esta solução pareça custosa, na implementação, o algoritmo só precisará criar um temporizador, diminuindo o número de *threads* criadas.

A associação de temporizadores às VCs, subexpressões ou expressões da regra, é feita de maneira simples, associando um símbolo à expressão da regra, contemplado na gramática.

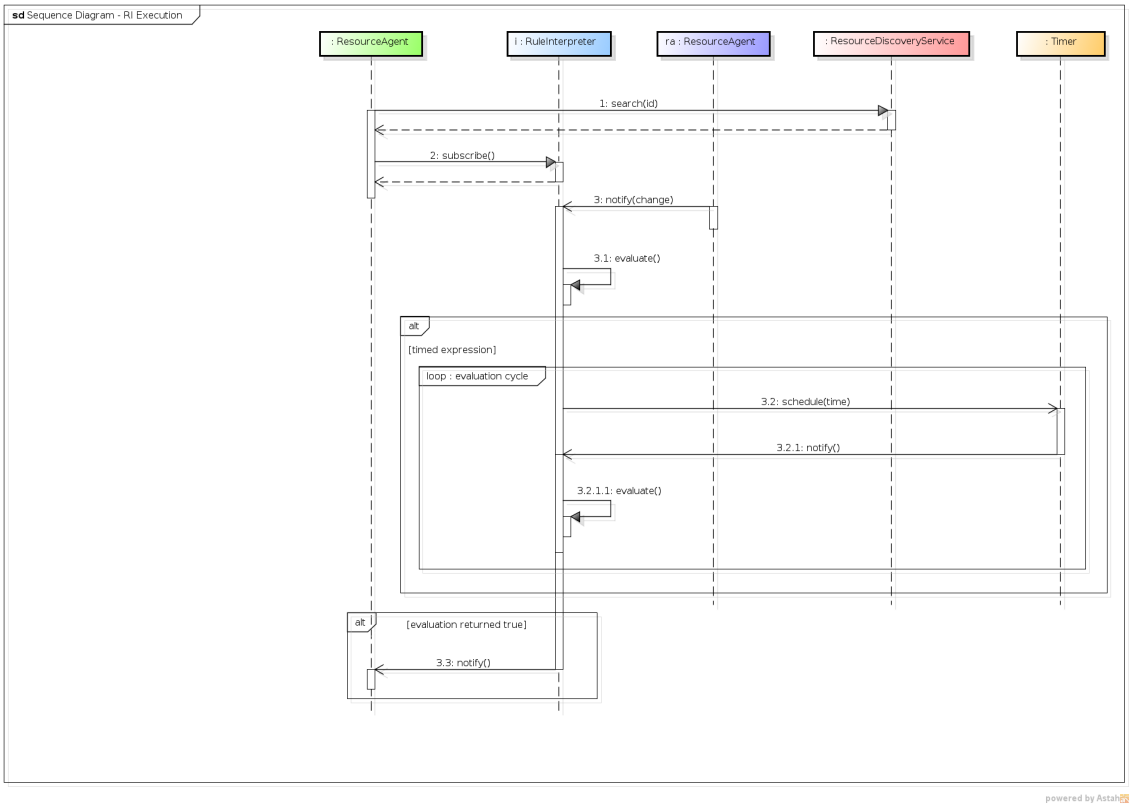

Figura 10. Interpretador de Regra em execução

5. Conclusão

Este trabalho procurou definir conceitos utilizados no ambiente do SmartAndroid, no que concerne ao contexto e a interação com o mesmo. Foi estudado como se expor as informações de contexto de um recurso assim como suas funcionalidades, ou serviços, e foi proposta uma implementação que possibilita essas declarações com um mínimo de esforço, do ponto de vista do programador.

A fim de se elevar a abstração em torno das variáveis de contexto foi proposto o Interpretador de Regra, um agente que inter-relaciona estas variáveis por meio de uma regra, capaz de ser criado em tempo de execução. As regras criadas podem também incluir o componente tempo, possibilitando que as variáveis de contexto sejam monitoradas, e não só avaliadas, o que aumenta o poder de expressão da regra.

A arquitetura descentralizada de manutenção do contexto e da interpretação do contexto, traz vantagens. Do ponto de vista de desempenho, embora seja comum a utilização de motores de regras – com algoritmos (como o RETE [Forgy 1982]) para a execução eficiente de um conjunto de regras –, os Interpretadores de Regras possuem a propriedade da execução em paralelo das regras, reduzindo o tempo, se comparado com uma execução sequencial. O fato de não se ter um contêiner de contexto centralizado diminui também o uso rede, e portanto o consumo de bateria dos dispositivos. Do ponto de vista de modelagem, os interpretadores oferecem uma interface de simples utilização para o programador e de fácil manutenção, por ser semelhante à utilização de um outro Agente de Recurso qualquer.

Referências

Abowd, G., Dey, A., Brown, P., Davies, N., Smith, M., and Steggles, P. (1999). Towards a better understanding of context and context-awareness. In *Handheld and Ubiquitous Computing*, pages 304–307. Springer.

Augusto, J. and McCullagh, P. (2007). Ambient intelligence: Concepts and applications. *Computer Science and Information Systems/ComSIS*, 4(1):1–26.

Bezerra, L. (2011). Uso de ontologia em serviço de contexto e descoberta de recursos para autoadaptação de sistemas. Master's thesis.

Cardoso, L. and Sztajnberg, A. (2006). Self-adaptive applications using ADL contracts. *Self-Managed Networks, Systems,*, pages 87–101.

Chen, G. and Kotz, D. (2002). Solar : An Open Platform for Context-Aware Mobile Applications. (June):41–47.

Chen, Y.S. and Chen, I.C. and Chang, W. (2010). Context-aware services based on OSGi for smart homes. *Ubi-media Computing (U-Media), 2010 3rd IEEE International Conference on*, 11:392.

Coulouris, G., Dollimore, J., and Kindberg, T. (2005). *Distributed systems: concepts and design*. Addison-Wesley Longman.

Dey, A., Abowd, G., and Salber, D. (2001). A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction*, 16(2):97–166.

Forgy, C. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37.

Lee, Y., Iyengar, S., Min, C., Ju, Y., Kang, S., Park, T., Lee, J., Rhee, Y., and Song, J. (2012). Mobicon: a mobile context-monitoring platform. *Communications of the ACM*, 55(3):54–65.

Liu, H. and Parashar, M. (2003). Dios++: A framework for rule-based autonomic management of distributed scientific applications. *Euro-Par 2003 Parallel Processing*, pages 66–73.

Lyytinen, K. and Yoo, Y. (2002). Ubiquitous computing. *COMMUNICATIONSOF THE ACM*, 45(12):63.

Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R., and Mickunas, M. (2005). Olympus: A high-level programming model for pervasive computing environments. In *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on*, pages 7–16. IEEE.

Saha, D. and Mukherjee, A. (2003). Pervasive computing: a paradigm for the 21st century. *Computer*, 36(3):25–31.

Satyanarayanan, M. (2001). Pervasive computing: Vision and challenges. *Personal Communications, IEEE*, 8(4):10–17.

Sudha, R., Rajagopalan, M., Selvanayaki, M., and Selvi, S. (2007). Ubiquitous semantic space: A context-aware and coordination middleware for ubiquitous computing.

In *Communication Systems Software and Middleware, 2007. COMSWARE 2007. 2nd International Conference on*, pages 1–7. IEEE.

Wang, Q. (2005). Towards a rule model for self-adaptive software. *ACM SIGSOFT Software Engineering Notes*, 30(1):8.

Weis, T., Knoll, M., and Ulbrich, A. (2007). Rapid prototyping for pervasive applications. *IEEE Pervasive*.

Weiser, M. (1991). The computer for the 21st century. *Scientific American*, 265(3):94–104.