Uma Estratégia para Integração de Restrições Arquiteturais em Design Rules

Éberton S. Marinho[1], Thaís Batista[1], Flávia C. Delicato[1], Paulo F. Pires[1]

[1]Programa de Pós-Graduação em Sistemas e Computação – Universidade Federal do Rio Grande do Norte (UFRN) Caixa Postal 1524 – 59072-970 – Natal – RN – Brazil

ebertonsm@gmail.com, {thais,flavia.delicato, paulo.pires}@dimap.ufrn.br

Abstract. *Architectural Description Languages (ADLs) specify elements that comprise software architecture, the relationship between such element, as well as architectural constraints. Representation of architectural constraints in design rules that guide the development process is necessary to enforce architectural properties. In this article, we propose a strategy for mapping architectural constraints described in AspectualACME to a design rules language semantically closer to Aspect Oriented languages, thus assuring the propagation of concepts defined at design phase to the remainder development phases. The article also includes a case study to exemplify the proposed strategy.*

Resumo. *Linguagens de Descrição Arquitetural (LDAs) especificam os elementos que fazem parte da arquitetura do software, a composição entre esses elementos, bem como restrições arquiteturais. A representação de restrições arquiteturais em design rules que norteiam o desenvolvimento é essencial para que propriedades arquiteturais sejam obedecidas. Neste artigo propomos uma estratégia para o mapeamento de restrições arquiteturais, especificadas em AspectualACME, para uma linguagem de design rules semanticamente mais próxima do formato adotado pelas linguagens de implementação orientadas a aspectos do que outras abordagens existentes, garantindo assim a devida propagação dos conceitos especificados em nível de projeto para os demais.O artigo também apresenta um estudo de caso que exemplifica a estratégia proposta.*

1. Introdução

Uma das fases mais importantes no desenvolvimento de software é a fase de projeto (*design*), onde se criam artefatos documentais da arquitetura do sistema para guiar as fases seguintes de desenvolvimento [Sommerville 2005]. Os objetivos dessa fase são especificar as abstrações do sistema e como elas se relacionam, bem como, estabelecer restrições sobre tais abstrações, norteando assim, o correto desenvolvimento do sistema segundo os requisitos mínimos acordados entre as partes.

Para se obter software com alto grau de qualidade, é necessário produzir, desde a fase de projeto, sistemas com alta modularidade. Tal característica é obtida com o

emprego de vários paradigmas e técnicas. Um paradigma emergente é o Desenvolvimento de Sistemas Orientados a Aspectos (DSOA) [Kiczales et al. 1997]. A principal motivação para o surgimento de DSOA foi a necessidade de separar conceitos básicos de conceitos transversais[1], que desfavorecem a modularidade e o reuso. Nessa abordagem, os conceitos transversais são encapsulados como aspectos, em tempo de projeto, e unidos aos conceitos básicos em tempo de compilação ou execução.

No entanto, estudos recentes [Sullivan et al. 2001, Sullivan et al. 2005, Griswold et al. 2006, Ribeiro et al. 2001] mostraram que a utilização da Programação Orientada a Aspectos (POA), apesar de ser um meio efetivo para modularização dos conceitos transversais, pode prejudicar a modularidade dos demais conceitos, principalmente porque as linguagens de programação atuais que dão suporte a POA, como AspectJ, encontram-se em um nível de abstração baixo [Kiczales et al. 2001], o que cria um alto acoplamento entre os conceitos transversais e os básicos, interceptados pelos primeiros. Esse acoplamento desfavorece ainda o paralelismo de desenvolvimento de ambos os conceitos, pois mudanças nos conceitos básicos do sistema podem interferir no correto funcionamento dos aspectos.

Para a descrição dos artefatos gerados na fase de projeto de um sistema, Baldwin e Clark [2000] propõem que sejam utilizadas *design rules* para descrever tanto estruturas quanto restrições arquiteturais. *Design Rules* definem contratos que devem ser obedecidos em todas as fases posteriores do ciclo de vida do processo de construção do software. Essa abordagem, além de promover a modularidade do sistema, favorece o paralelismo de desenvolvimento.

Design rules podem ser formalmente expressas através do emprego de Linguagens de Descrição Arquitetural (LDAs). ACME [Garlan et al. 1997] é uma LDA que tem sido bastante referenciada na literatura e que propõe os seguintes conceitos: *Systems, Components, Ports, Roles, Attachments, Representations* e *Properties,* para definir a estrutura e o comportamento de um sistema. ACME inclui uma extensão, Armani [Monroe 2001], para descrição de restrições arquiteturais. Tais restrições são formadas por termos declarativos cuja sintaxe e semântica são baseadas na lógica de predicados de primeira ordem que podem fazer restrições tanto a estruturas como a comportamentos do sistema e funcionam como regras explícitas que devem ser estritamente cumpridas. AspectualACME [Batista et al. 2006] estende ACME adicionando estruturas para dar suporte ao DSOA.

Apesar dos benefícios advindos da utilização de *design rules* na fase de projeto, para que estes sejam propagados para as demais fases de desenvolvimento de sistemas, é necessário que tais *design rules* sejam concretizadas no código que implementa o sistema. Contudo, existe uma considerável distância entre as abstrações usadas nas LDAs atuais para descrição de sistema e aquelas usadas para implementá-las seguindo, por exemplo, o paradigma de Orientação a Objetos e o paradigma de Orientação a Aspectos [Goulo e Abreu 2003]. Na tentativa de aproximar as abstrações utilizadas para descrições arquiteturais de sistemas e linguagens de programação, há em Dósea et al. [2007] uma proposta de tradução de *design rules* expressas em AspectualACME para uma representação intermediária bem mais próxima do modelo POA geralmente utilizado por desenvolvedores de software. No entanto, em Dósea et al. [2007] o

[1] Conceitos transversais são conceitos que ficam espalhados e entrelaçados no código da aplicação e que não fazem parte da lógica de negócio principal do sistema.

mapeamento proposto leva em conta apenas *design rules* que representam os elementos estruturais da arquitetura, não sendo tratado o mapeamento das restrições arquiteturais definidas através da linguagem Armani. Dessa forma, as representações de *design rules* produzidas pelo mapeamento proposto por Dósea et al. [2007] não refletem de forma completa as descrições arquiteturais, podendo desrespeitar importantes restrições arquiteturais.

Este artigo tem como objetivo propor uma estratégia para o mapeamento de restrições arquiteturais especificadas em AspectualACME para uma linguagem de *design rules* mais próxima do código de implementação, a fim de diminuir a distância semântica entre restrições arquiteturais e implementação, garantindo assim a devida propagação dos conceitos especificados em nível de projeto para as demais fases de desenvolvimento. Para isso, estendemos a linguagem proposta em Dósea et al. [2007] adicionando a definição de uma linguagem de restrições arquiteturais e definindo regras de tradução para dar suporte à extensão proposta. O artigo inclui também um estudo de caso para melhor ilustrar a estratégia proposta.

Este artigo está estruturado da seguinte forma: a Seção 2 apresenta sucintamente os conceitos básicos sobre AspectualACME, Armani, e sobre *design rules*. A Seção 3 apresenta a linguagem para restrições arquiteturais proposta nesse artigo juntamente com suas regras de mapeamento. A Seção 4 descreve o estudo de caso realizado para ilustrar a proposta. A Seção 5 descreve os trabalhos relacionados. A Seção 6 contém as conclusões e possibilidades de trabalhos futuros.

2. Conceitos Básicos

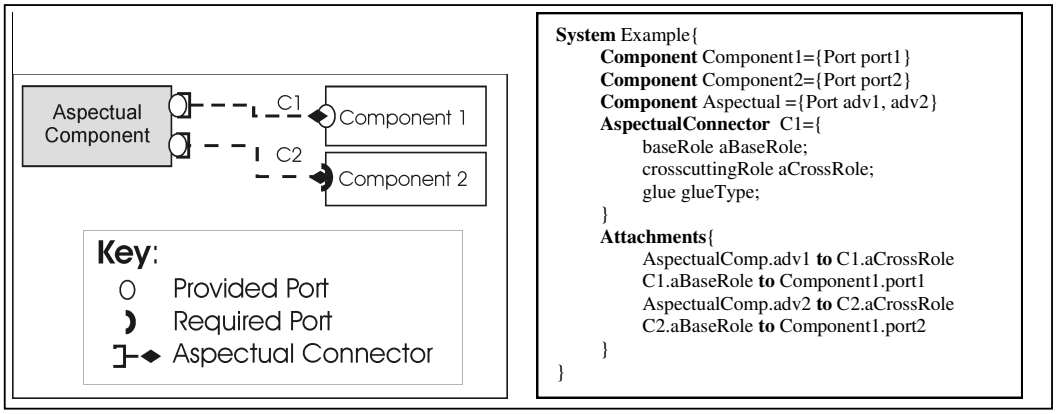
2.1. AspectualACME

O DSOA propõe um conjunto de conceitos e técnicas para tratar os conceitos transversais que geralmente estão espalhados e entrelaçados no código da aplicação. Tais conceitos são encapsulados em uma unidade de modularização, o aspecto, e mecanismos são usados para composição dos aspectos com os elementos base, que descrevem a lógica da aplicação. Os *joinpoints* são pontos nos elementos base que são interceptados pelos aspectos. Expressões de *pointcuts* permitem a seleção do conjunto de pontos de composição (*joinpoints*) para um determinado aspecto. *Advices* indicam o comportamento que será inserido nos *joinpoints*. Eles possuem duas partes, a primeira é o *pointcut,* que determina as regras de captura dos *joinpoints;* a segunda é o código que será executado quando ocorrer o ponto de junção definido pela primeira parte. Cada *advice* possui um tipo (*after, before e around)* que descreve quando o comportamento deve ser executado nos *joinpoints. inter-type declarations* adicionam características (por exemplo, comportamentos e atributos) nos pontos interceptados.

Os conceitos base e os aspectos são modelados como componentes em AspectualACME. O ponto central de AspectualACME baseia-se em mecanismos de composição construídos em torno do *Conector Aspectual (AC)* [Garcia et al. 2006] que é uma extensão dos conectores tradicionais para descrever interações entre conceitos transversais e componentes. O *Conector Aspectual* possui uma interface com três partes: (1) um *base role*, (2) um *crosscutting role* e (3) uma cláusula *glue*. O *croscutting role* deve ser conectado a uma porta do componente que representa o conceito transversal. O *base role* deve ser conectado a(s) porta(s) do(s) componente(s) que representa(m) o(s) conceito(s) básico(s) a ser(em) alvo do conceito transversal. A

cláusula *glue* especifica detalhes sobre a interceptação, como por exemplo, quando o *advice* deve afetar o componente – *after, before, around* e outros.

A Figura 1 (a) mostra graficamente um exemplo que utiliza um Componente "Aspectual" (denominação usada apenas para distinção na figura) que afeta dois componentes base (Component 1 e Component 2). Na Figura 1 (b) parte de seu respectivo código fonte é mostrado.



(a) (b)

Figura 1. (a) Ilustração da interceptação de componentes por um Aspectual Component. (b) código fonte referente à descrição gráfica em (a).

Em AspectualACME restrições arquiteturais são definidas usando uma linguagem baseada na lógica de predicados de primeira ordem, a linguagem Armani. As restrições que são permitidas em Armani podem ser do tipo *Invariant* ou *Heuristic*. O *invariant* possui um predicado com uma condição que deve ser satisfeita pela descrição arquitetural no escopo onde a restrição se encontra. A *Heuristic* tem o significado semântico de um "conselho" que seria importante que o sistema seguisse, porém não é obrigatório como o *invariant*.

Armani aceita que quantificações sobre variáveis sejam feitas apenas sobre conjuntos finitos. Um exemplo de restrições arquiteturais é mostrado na Figura 2 onde há três *invariants* e um *heuristic*. No primeiro *invariant* há uma declaração de que para todo *p* pertencente ao conjunto de portas (*Ports)* do componente *Client*, o protocolo desta porta deve ser inicializado com o objeto *rpc-client*. O segundo *invariant* restringe uma característica estrutural do componente: ele não pode ter mais que cinco portas. O terceiro *invariant* restringe um aspecto dinâmico do componente: para os valores que *request-rate* venha a assumir, ele deve sempre ser maior ou igual à zero (0). A heurística "*Heuristic request-rate < 100"* indica que a taxa de requisição deve ser menor que cem (100) para que se tenha o provimento de um serviço com qualidade, por exemplo.

Armani dá suporte à definição de tipos básicos como: int, float, string e booleano, mas também de tipos mais complexos no formato de conjuntos como: *set, sequence, enum* e *record*. A linguagem de predicados de Armani conta com uma API com definição de: funções para verificação de tipos, funções para conectividade de grafos, funções de parentesco e funções de conjuntos. Além da possibilidade de poder definir variáveis e constantes.

```
Component Type Client = {
        Port Request = { Property protocol : CSProtocolT = rpc-client };
        Property request-rate : float << default = 0.0 >>;
        Invariant forall p in self.Ports • p.protocol = rpc-client;
        Invariant size(self.Ports) <= 5;
        Invariant request-rate >= 0;
        Heuristic request-rate < 100;
}
```

Figura 2. Exemplo de uma restrição arquitetural em Armani.

Os operadores oferecidos por Armani são operadores de comparação, operadores lógicos e operadores aritméticos, além dos quantificadores existencial (*exists*) e universal (*forall*). A linguagem também fornece funções *select* e *collect*, cujo valor semântico é o de seleção em um conjunto sob um determinado critério. A diferença entre as duas é que *select* retorna um subconjunto do conjunto declarado que satisfaz ao predicado especificado, enquanto que *collect* retorna um conjunto dos valores que estão associados a cada elemento que satisfaz o predicado.

2.3 Design Rules

Segundo Ribeiro et al. [2007], DSOA consegue apenas agregar a implementação dos conceitos transversais obtendo, dessa forma, a modularidade desses conceitos. No entanto, as construções que dão suporte ao DSOA encontram-se em um nível de abstração onde há um forte acoplamento entre a classe e o aspecto, o que faz com que qualquer mudança na implementação da classe possa comprometer o correto funcionamento do aspecto construído. Tal fato prejudica a modularidade das classes, pois a implementação da classe não pode ser compreendida ou alterada sem que o desenvolvedor da classe tenha consciência dos aspectos que podem interceptá-las. Também não é possível desenvolver as classes paralelamente aos aspectos visto que os aspectos são altamente dependentes dos detalhes de implementação das classes.

Em Baldwin e Clark [2000] há a definição de *design rules* como uma camada de abstração acima da camada de implementação. No contexto de DSOA, o uso de tais regras amenizaria as dependências entre os conceitos básicos e transversais ao estabelecer um contrato que deve ser rigidamente seguido por ambos. Uma ilustração dessa idéia é mostrada na Figura 3 onde a *Regra de Design* estabelece que "Todo método da classe *GUI* será distribuído pelo aspecto Distribution".

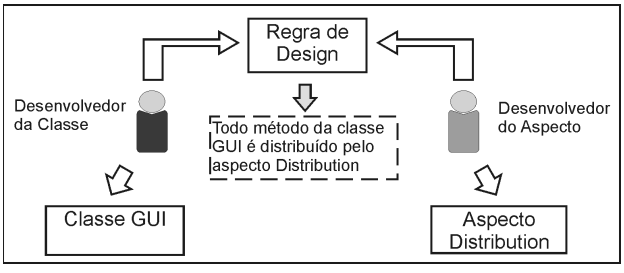


Figura 3. Exemplo de Design Rule.

Em Dósea et al. [2007] há uma proposta de mapeamento da LDA AspectualACME para definições de *design rules*, na tentativa de amenizar o problema da distância semântica entre LDAs e linguagens de programação, ao mesmo tempo em que favorece a modularização de classes e de conceitos transversais, o reuso de módulos

do sistema e o paralelismo na fase de desenvolvimento, ao criar uma definição clara e mínima, na fase de projeto (um contrato), das interfaces das classes e aspectos que farão parte do sistema.

A Figura 4 mostra o meta-modelo da linguagem proposta em Dósea et al. [2007] para representar *Design rules*. Na Figura 4 (a) a descrição de uma regra é composta por uma ou mais *regras estruturais*, que são utilizadas para definir regras que classes ou aspectos devem obedecer. Classes ou aspectos podem, ainda, definir atributos, assinaturas de métodos e *regras comportamentais*. Tais regras podem ser definidas dentro do escopo de classes e aspectos ou dentro do escopo de métodos adendos (*advices*). Pode ainda existir um módulo de configuração, como ilustrado na Figura 4 (b), usado para descrever uma opção de configuração para as *regras estruturais* descritas por uma regra de design. O módulo de configuração indica quais componentes serão utilizados por uma certa instância desse sistema.

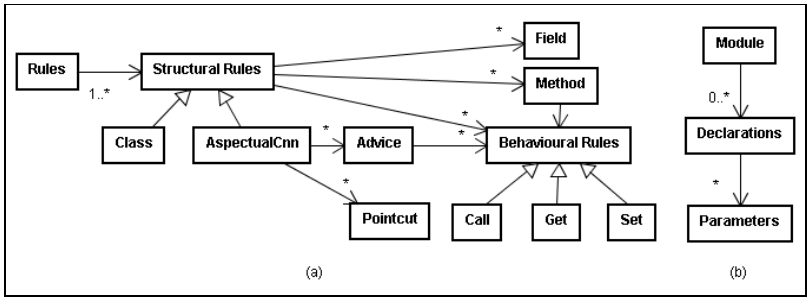


Figura 4. Meta-modelo de Design Rule.

A Tabela 1 mostra as regras comportamentais suportadas atualmente pela linguagem e que podem ser negadas através do operador !.

Tabela 1. Tabela com as regras comportamentais suportadas atualmente pela linguagem de *Design Rules proposta em Dósea et al. [2007]*

Regra	Descrição
call(método)	Deve existir uma chamada ao método passado como parâmetro dentro do escopo definido.
xcall(método)	Deve existir uma chamada ao método passado como parâmetro exclusivamente dentro do escopo definido.
set(método)	Deve existir uma mudança de estado do atributo passado como parâmetro dentro do escopo definido.
xset(método)	Deve existir uma mudança de estado do atributo passado como parâmetro exclusivamente dentro do escopo definido.
get(método)	Deve existir uma leitura do atributo passado como parâmetro dentro do escopo definido.
xget(método)	Deve existir uma leitura do atributo passado como parâmetro exclusivamente dentro do escopo definido

Dósea et al. [2007] definem as seguintes regras de transformação de AspectualACME para *design rules*: os componentes são mapeados em classes (*classes*), as propriedades em atributos (*fields*) e as portas em métodos (*methods*); os relacionamentos realizados pelos conectores são transformados em dependências entre classes e detalhados pelas regras comportamentais da Tabela 1; os Conectores Aspectuais são transformados na *strutural rule aspectualCnn;* a porta ligada a *baseRole* na cláusula *attachments* é transformada no *pointcut* do *aspectualCnn*; a cláusula *Glue* é mapeada para a parte do *advice* que especifica quando o conceito transversal intercepta o conceito base.

3 Abordagem para o Mapeamento de Restrições Arquiteturais

As seções anteriores mostraram a importância de se usar *design rules* na fase de projeto, bem como uma proposta descrita em Dósea et al. [2007], onde se define um conjunto de regras de mapeamento de AspectualACME para uma linguagem de representação de *design rules* mais próxima do modelo utilizado pela POA. No entanto, tais regras ainda não contemplam restrições arquiteturais. As subseções a seguir mostrarão: a proposta do presente trabalho para uma extensão do meta-modelo proposto em Dósea et al. [2007] para dar suporte a restrições arquiteturais em AspectualACME (Subseção 3.1); a sintaxe da linguagem de restrições arquiteturais no formato de *design rules* proposto por este trabalho (Subseção 3.2) e as regras de transformação para fazer o mapeamento da LDA Armani para tais *design rules* (Subseção 3.3).

3.1 Meta-modelo da Linguagem de Restrições Arquiteturais

A Figura 5 mostra o conjunto de componentes adicionados à linguagem de *design rules* proposta em [Dósea et al. 2007]. Tal extensão é evidenciada na Figura 5 através do agrupamento denominado "*DR Architectural Constraints*". A Figura 5 mostra também como se dá o relacionamento entre as restrições arquiteturais e os elementos da arquitetura estendida. Uma *Rule* pode ter inúmeras *Constraints,* as quais podem restringir estruturas ou comportamentos das *structural rules*. Por exemplo, poderia haver uma restrição quanto ao número máximo de métodos que uma classe poderia ter ou haver uma restrição que um determinado atributo deveria ter um valor dentro de uma faixa previamente definida.

Como mencionado na Subseção 2.2, existem dois tipos de restrições*: Invariant* e *Heuristic*. Na linguagem de *design rules*, tais cláusulas encapsulam um predicado (*predicate*) que pode limitar estruturas e comportamentos de um sistema.

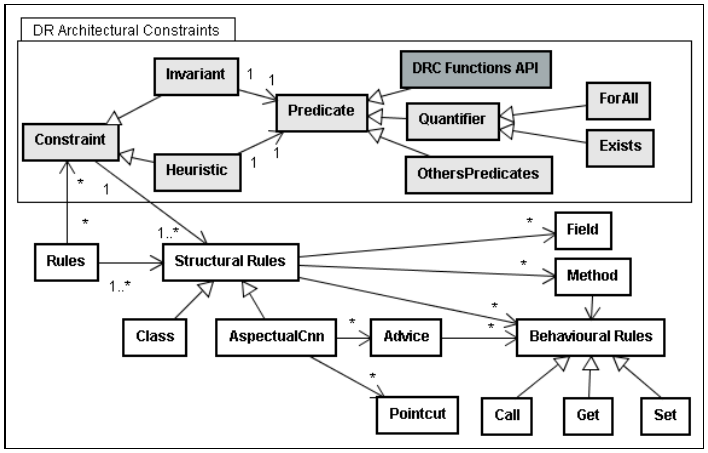


Figura 5. Ilustração do Meta-modelo estendido para suportar restrições arquiteturais.

A linguagem Armani possui uma série de funções, como funções para verificação de tipo de um elemento, funções de conjuntos, etc., para dar suporte a sua linguagem de predicados. A finalidade da *DRC Functions API* (classe em maior destaque na Figura 5) é ser um repositório de funções disponíveis na linguagem Armani e outras definidas pelo usuário. Por exemplo, na linguagem de predicados Armani a

seguinte restrição "*satisfiesType(e : elementType)*", que retorna um valor booleano, possui uma função equivalente na *DRC Functions API.*

3.2 A Sintaxe da Linguagem de Restrições Arquiteturais

A linguagem de *design rules* para restrições arquiteturais proposta no presente trabalho estabelece que, inicialmente, toda restrição deve ser apresentada em uma cláusula *invariant* ou *heuristic*. A cláusula *invariant* contém um predicado que deve ser estritamente obedecido pelos elementos aos quais o predicado faz referência. A cláusula *heuristic* funciona como um conselho que seria interessante que fosse cumprido pelos elementos referenciados pelo predicado da *heuristic*. O predicado pode conter: funções definidas na *DRC Functions API*; operadores de comparação, lógicos, aritméticos e de atribuição; conjuntos; identificadores e constantes.

Como visto na subseção anterior, *DRC Functions API* funciona como um repositório de funções baseadas nas funções do Armani, onde se podem encontrar: Funções para suporte a verificação e manipulação de tipos; Funções de suporte a operações de conectividade de grafos; Funções para manipulação de parentesco; Funções de suporte a operações com conjuntos; Funções lógicas; Funções aritméticas; Funções de quantificação; Funções de seleção.

Além das funções anteriormente citadas, a linguagem de *design rules* também dá suporte aos operadores de:

- Comparação: <, <=, >, >=, apenas para números e ==, != para números e objetos
- Lógicos: && (e lógico), || (ou lógico), !(negação) e as funções que ficam no DRC Functions API: fimp, que é equivalente ao operador de implicação; fequiv, que é equivalente ao operador de equivalência; fxor, que é equivalente ao operador de ou exclusivo.
- Aritméticos: +, -, *, / com números e a função fmod, para o operador módulo
- Atribuição: =, cujo significado semântico é que a variável do lado esquerdo do operador deve ser inicializada com o valor presente no lado direito do operador.

A linguagem de *design rules* para restrições arquiteturais também dá suporte à definição de conjuntos. Os conjuntos suportados pela linguagem de restrição e uma breve descrição de cada uma são apresentados a seguir:

- *Record:* aceita conjuntos de pares ordenados (nome, valor), que guarda elementos com nomes diferentes.
- *Sequence:* lista ordenada de elementos que podem ser repetidos.
- *Set:* define uma lista de elementos que não podem ser repetidos.

Para dar maior expressividade à linguagem de descrição arquitetural, também são oferecidas funções com valor semântico dos quantificadores universal e existencial. Ambos os quantificadores recebem três parâmetros, como mostrado na sintaxe abaixo:

- Quantificador universal: *<TBoolean>* ***forall*** *(<TSetType{(<TIdentifier>)*}>, <TSetType {(<TElement>)*}>, <TPredicate>)*
- Quantificador existencial: *<TBoolean>* ***exists*** *(<TSetType {(<TIdentifier>)*}>, < TSetType {(<TElement>)*}>, <TPredicate>)*

O primeiro parâmetro de um quantificador é um conjunto de identificadores, os quais serão valorados com o conjunto de elementos do segundo parâmetro. O terceiro

parâmetro é o predicado que precisa ser satisfeito para todas as valorações que o conjunto de identificadores possa assumir, para o caso da função *forall*, ou ter pelo menos uma valoração que satisfaça o predicado, no caso da função *exists*.

3.3 Regras de Transformação

Esta seção descreve o conjunto de regras de transformações utilizadas para mapear restrições arquiteturais descritas em AspectualACME para *design rules*. Como já mencionado na subseção 2.2, a linguagem Armani dispõe de um conjunto de funções que auxiliam na tarefa de restringir características de sistemas. Para o mapeamento entre essas funções e a linguagem de regras de design, foi criado o repositório de funções *DRC Functions API* que contém funções equivalentes as da linguagem Armani em *design rules*. Ou seja, para toda função utilizada em Armani existe uma correspondente em *design rules*. A sintaxe das funções da *DRC Functions API* é: *<ReturnType> <FunctionName> "(" [(<Type>)*] ")"*

Além das funções existentes em Armani, foram adicionadas outras correspondentes a:

- *Funções lógicas*: que em Armani são representadas por operadores, e na linguagem de *design rules* assumem a forma de funções com a seguinte correspondência: os operadores em Armani: →, ↔, xor, são descritos em *design rules* como: fimp, fequiv, fxor.
- *Função aritmética mod*: que em Armani é representado pelo operador mod, na linguagem de *design rules* corresponde à função fmod.
- *Funções de quantificação*:
    - Em Armani os quantificadores são expressos por fórmulas lógicas do formato: ("forall" | "exists")? <identifier>[("," <identifier>)*] "in" <SetExpression> "|" <Predicate>
    - Em *design rules* os mesmos quantificadores são expressos por funções que recebem três parâmetros: um conjunto de identificadores; um conjunto com os quais os identificadores devem ser valorados e um predicado que vai avaliar as quantificações das variáveis.
- Funções de seleção: para se fazer uma seleção em um conjunto, obedecendo a um determinado critério, Armani dispõe das expressões *select* e *collect*. Na linguagem de *design rules* tais declarações foram traduzidas para funções (*fselect* e *fcollect*) que recebem três parâmetros: um conjunto de variáveis que serão valoradas com o conjunto passado como segundo parâmetro, e um predicado que indica quais restrições o subconjunto de retorno deve satisfazer.

4 Estudo de Caso

Com a finalidade de exemplificar a proposta apresentada no presente trabalho, foi criado um estudo de caso baseado no Sistema *Health Watcher(SHW)* que é um exemplo clássico e amplamente referenciado na literatura de DSOA. As subseções a seguir apresentarão a contextualização e a arquitetura do *SHW* descritas em AspectualACME e, em seguida, a aplicação de Restrições Arquiteturais especificadas em AspectualACME ao *SHW*. A última subseção trata da tradução de tais restrições para *design rules*.

4.1 Arquitetura do Sistema *Health Watcher (SHW)*

O SHW é um sistema de informações para Web desenvolvido pelo grupo de pesquisa *Software Productivity* da Universidade de Pernambuco [SPG 2006]. Tal sistema suporta o registro de reclamações para o Sistema Público de Saúde. A Figura 6 mostra a descrição arquitetural do sistema. O componente *GUI* provê uma interface gráfica Web de comunicação entre o sistema e o usuário. O componente *Business* encapsula as regras de negócios da aplicação. O componente *Data* é encarregado de efetivar o registro de informações do sistema em um meio de persistência. A Figura 6 mostra um dos conceitos transversais desse sistema, a Acessibilidade (*Accessability*). Tal conceito irá interceptar os pontos onde há a necessidade de um controle de acesso aos recursos do sistema.

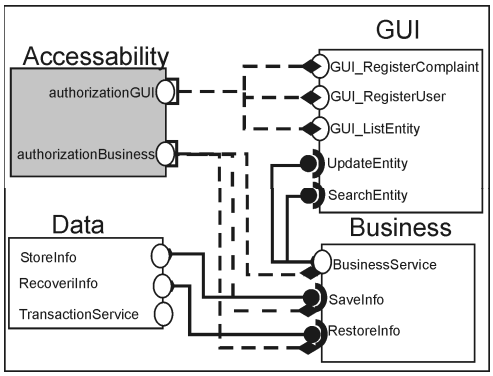


Figura 6. Ilustração da descrição arquitetural do Sistema Health Watcher.

Como se pode observar na Figura 6, o conceito transversal *Accessability* intercepta todos os recursos com os quais o usuário pode interagir na interface gráfica (*GUI*) e na camada de regras de negócios (*Business*). Mais especificamente, o conceito transversal *Accessability* intercepta todas as portas com prefixo "*GUI*" do componente *GUI* e todas as portas do componente *Business*. Tais generalizações são feitas no código através do quantificador "*", na parte de configuração da descrição arquitetural (*Attachments*).

```
Component GUI = {
   Port GUI_RegisterComplaint;
   Port GUI_RegisterUser;
   Port GUI_ListEntity;
   Port UpdateEntity;
   Port SearchEntity;
}
Component Accessability={
Port authorizationGUI;
Port authorizationBusiness;
}
AspectualConnector AccessCnnGUI={
   BaseRole sink;
   CrosscuttingRole source;
   glue source before sink;
}
```

(a)

```
Attachments
Accessability.authorizationGUI to
AccessCnnGUI.source;

AccessCnnGUI.sink to GUI.GUI*;

Accessability.authorizationBusiness to
AccessCnnBusiness.source;

accessCnnBusiness.sink to
Business.storeInfo;
```

(b)

Figura 7. Parte do código em AspectualACME correspondente ao exemplo da Figura 6.

A Figura 7 (a) mostra o código fonte dessa descrição feita em AspectualACME, onde o componente *Accessability* e o componente *Business* são conectados ao Conector

Aspectual *AccessCnnBusiness*. A declaração dessa conexão é feita na parte de configuração (*Attachments*) da descrição arquitetural ilustrada na Figura 7(b). Ainda na Figura 7(a), pode-se observar a declaração *glue,* a qual indica de que forma a *CrosscuttingRole* e a *BaseRole* estão relacionadas.

4.2 Aplicação de Restrições Arquiteturais ao Sistema Health Watcher

Essa Subseção mostra um conjunto de regras estruturais que podem ser inseridas à descrição arquitetural com a finalidade de garantir que a arquitetura atual ou futuras evoluções do sistema não entrem em desacordo com os requisitos básicos que o sistema deve possuir. Como descrito na seção anterior, o conceito transversal *Accessability* deve interceptar todos os recursos nos quais haja a necessidade de verificação de permissão. Uma forma de garantir que isso aconteça é criando uma restrição arquitetural que descreva essa informação. Uma possível restrição arquitetural de escopo global que faz essa restrição é a seguinte:

```
Invariant forall res in {select comp in HealthWatcher.Components|
   declaredSubType(comp, GUI) || declaredSubType(comp, Business)}|
      connected(Accessability, res);
```

Essa restrição declara que para toda variável *res*, que seja do tipo *GUI* ou *Business*, *Accessability* deve estar conectado a tal(is) componente(s). O poder semântico dessa declaração é suficientemente expressivo para garantir que todos os componentes do tipo *GUI* ou *Business* seja interceptado pelo *Accessability*.

No entanto, pode-se restringir mais ainda a declaração anterior adicionando os seguintes *invariants* ao escopo do Conector Aspectual *AccessCnnGUI*:

```
Invariant attached(source, AccessCnnGUI.authorizationGUI);
Invariant forall gui in {select comp in HealthWatcher.Components |
   declaredSubType(comp, GUI)}|
      forall p in {select pts in gui.Ports}|attached(sink,p);
```

O primeiro invariant é uma restrição a qual especifica que o *Role source* do Conector Aspectual deve estar sempre conectado a porta *authorizationGUI* do componente *Accessability*. O segundo *invariant* descreve que para todas as portas de todos os componentes do tipo *GUI*, as portas desses componentes devem estar ligadas ao Conector Aspectual *AccessCnnGUI* através do *BaseRole sink.*

4.3 Tradução de Restrições Arquiteturais para *Design Rules*

Nessa subseção serão feitos os mapeamentos entre as restrições arquiteturais descritas na subseção anterior para *design rules* obedecendo às regras de transformações descritas na Subseção 3.3.

A Figura 8 mostra o resultado da aplicação das regras de transformações de restrições arquiteturais do SHW em *design rules*. Caso esse mapeamento não fosse realizado as restrições arquiteturais não seriam propagadas, gerando assim uma inconsistência entre a arquitetura e a implementação. Além disso, restrições arquiteturais no formato de *design rules* guiam o desenvolvimento, impedindo que os desenvolvedores violem as regras do sistema.

Como descrito na Subseção 2.3, a porta ligada ao *baseRole* é mapeada para um *pointcut* e a cláusula *glue* é transcrita para um *advice* que especifica o momento no qual a porta relacionada ao *crosscuttingRole* intercepta o conceito base. Na Figura 8, a linha

1 em destaque mostra a função *attached* que em AspectualACME receberia os parâmetros *port* e *role*, mas que em design rules recebe o *aspectualCnn AccessCnnGUI* e a assinatura do método *authorizationGUI* e verifica se há algum *advice* que chame esse método. Na linha 2, a função *attached* tem uma assinatura diferente e verificar se existe em AccessCnnGUI algum conceito transversal que intercepte método *p* antes de sua execução.

```
dr HealthWatcher{
   Invariant forall({res}, select({comp}, getElements(HealthWatcher),
      declaredSubType(comp, GUI) || declaredSubType(comp, Business)),
         connected(Accessability, res));

❶  Invariant attached(AccessCnnGUI, authorizationGUI);
   Invariant forall({gui}, select({comp}, getElements(HealthWatcher),
      declaredSubType(comp, GUI)), forall({p}, getMethods(gui),
❷        attached(AccessCnnGUI, {{before, p}}));

   Invariant attached(AccessCnnBusiness, authorizationBusiness);
   Invariant forall({business}, select({comp}, getElements(HealthWatcher),
       declaredSubType(comp, Business)), forall({p}, getMethods(business),
             attached(AccessCnnBusiness, {{before, p}}));
}
```

Figura 8. Resultado da aplicação das regras de mapeamentos de restrições arquiteturais para *design rules*.

5 Trabalhos Relacionados

O uso de *design rules* tem sido discutido em outros trabalhos. Por exemplo, em Sullivan et al. [2005] é utilizada uma abordagem que usa linguagem natural para especificar *design rules*. No entanto, tal abordagem é imprecisa para garantir e verificar as regras estabelecidas. Em Chavez et al. [2006] propõe-se um modelo visual que expressa alguns tipos de *design rules*. Apesar de uma especificação menos ambígua, o modelo não permite a verificação automática do código. Por outro lado, a abordagem proposta no presente trabalho tem como base a linguagem de lógica de primeira ordem Armani, o que reduz drasticamente a imprecisão e ambigüidade, características intrínsecas da linguagem natural.

Em Griswold et al. [2006] é proposta uma abordagem baseada em contrato Meyer [1992], a qual utiliza construções em AspectJ para criar invariantes em forma de interfaces que fazem restrições a estruturas e comportamentos de suas implementações. Essa interface com código de restrições é chamada de XPI (*crosscut programming interfaces*). No entanto, tentar especificar *design rules* utilizando AspectJ resulta em um complexo mecanismo de verificação, difícil de descrever e entender devido à complexidade das construções necessárias para verificação de tais regras. O trabalho apresentado no presente trabalho utiliza uma linguagem de alto nível para realizar essa tarefa, o que diminui consideravelmente a complexidade para descrição de *design rules*, sem entretanto, perder o poder de expressão necessário pra realizar tal feito. Além disso, este artigo trata tais conceitos no nível de projeto, enquanto que Griswold et al. [2006] se refere ao nível de implementação.

Em Morgan et al. [2007], apresenta-se uma linguagem inspirada pela linguagem de pointcut em AspectJ, para declarativamente codificar *design rules* para sistemas OO. Entretanto, esta linguagem serve apenas para sistemas OO e não permite uma definição clara de regras para propiciar o desenvolvimento paralelo dos componentes do software.

A estratégia apresentada aqui aborda os paradigmas OO e AO, e ainda permite o desenvolvimento paralelo dos conceitos OO e AO na fase de desenvolvimento.

6 Conclusões

Este artigo apresentou um conjunto de regras de mapeamento para transformar restrições arquiteturais especificadas em AspectualACME para um formato semanticamente mais próximo do adotado pelas linguagens de implementação orientadas a aspectos do que outras abordagens existentes. Também apresentou uma linguagem para descrição de tais restrições arquiteturais concretizando as regras especificadas. As transformações propostas estendem e complementam o trabalho desenvolvido em Dósea et al. [2007], a fim de efetivar as contribuições daquele trabalho, ao garantir que a implementação do sistema leve em conta as restrições arquiteturais, evitando inconsistências entre a descrição arquitetural e a implementação.

Além disso, as regras propostas promovem o paralelismo ao especificar, ainda na fase de projeto, as regras principais que desenvolvedores devem seguir e os limites que não podem ser violados pelo sistema. Como as restrições permitidas são apenas sobre conjuntos finitos, há ainda a possibilidade de verificar se o sistema não viola tais regras, o que dá uma maior confiabilidade ao software desenvolvido.

Como trabalhos futuros pretende-se criar uma ferramenta para gerar *design rules* a partir de restrições arquiteturais em Armani. Tal ferramenta deve ainda verificar o código da implementação de sistemas a fim de verificar se o mesmo não viola as regras estáticas especificadas. Além disso, a mesma ferramenta deve gerar código seguindo POA, para interceptar características e comportamentos do sistema sob os quais as regras fazem restrições, a fim de garantir que o sistema não as violará em tempo de execução. Pretende-se ainda realizar análises comparativas de tempo de desenvolvimento, facilidade de uso, segurança, e outras métricas cabíveis, entre as abordagens de desenvolvimento utilizando restrições arquiteturais em forma de *design rules* e sem utilizar tais regras.

Referências Bibliográficas

Baldwin, C. Y. e Clark, K. B. (2000). Design Rules: The Power of Modularity, volume 1. The MIT Press, 1° edição.

Batista, T., Flach, C., Garcia, A., Sant´anna, C., Kulesza, U., e Lucena, C. (2006). Aspectual connectors: Supporting the Seamless Integration of Aspects and LDAs. XX Simpósio Brasileiro de Engenharia de Software (SBES 2006), pag. 17–32. Florianópolis–SC.

Chavez, C., Garcia, A., Kulesza, U., SantAnna, C., e de Lucena, C. J. P. (2006). Crosscutting Interfaces for Aspect-Oriented Modeling. Brazilian Computer Society, 12:43–58.

Dósea, M., Neto, A. C., Borba, P., e Soares, S. (2007). Specifying Design Rules in Aspect-Oriented Systems. I Latin American Workshop on Aspect-Oriented Software Development (LA-WASP'2007). evento realizado junto com SBES'07, João Pessoa-Brazil.

Garcia, A., Flach, C., Batista, T., Sant´anna, C., Kulesza, U., Rashid, A., e Lucena, C. (2006). On the Modular Representation of Architectural Aspects. European Workshop on Software Architecture (EWSA 2006), pag. 82–97. Nantes–FR.

Garlan, D., Monroe, R., e Wile, D. (1997). Acme: An Architecture Description Interchange Language. CASCON '97.

Goulo, M. e Abreu, F. (2003). Bridging the Gap Between ACME e Uml for CBD. Specification and Verification of Component-Based Systems (SAVCBS'2003).

Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., e Rajan, H. (2006). Modular Software Design with Crosscutting Interfaces. IEEE Software, pag. 51–60.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., e Griswold,W. G. (2001). Getting Started with Aspectj. Communications of the ACM, pag. 59–65.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., e Irwin, J. (1997). Aspect-Oriented Programming. European Conference on Object-Oriented Programming (ECOOP), pag. 220–242. Finland.

Meyer, B. (1992). Applying design by contract. IEEE Computer, pages 40–52.

Monroe, R. (2001). Capturing software architecture design expertise with armani. Technical report, Carnegie Mellon University.

Morgan, C., Volder, K. D., e Wohlstadter, E. (2007). A Static Aspect Language for Checking Design Rules. Proceedings of the 6th International Conference on Aspectoriented Software Development (DSOA '07), 12:63–72. New York–USA.

SPG - Software Productivity Group at UFPE. Disponível em: http://twiki.cin.ufpe.br/twiki/bin/view/SPG. 20 mai 2008.

Ribeiro, M., Dósea, M., Bonifácio, R., Neto, A. C., Borba, P., e Soares, S. (2007). Analyzing Class and Crosscutting Modularity Structure Matrixes. 21th Brazilian Symposium on Software Engineering (SBES 2007), pag. 51–60.

Soares, S., Laureano, E., and Borba, P. (2002). Implementing Distribution and Persistence Aspects with Aspectj. 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'2002), pag. 174–190.

Sommerville, I. (2005). Engenharia de Software. Addison Wesley, 6 edition.

Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., e Rajan, H. (2005). Information Hiding Interfaces for Aspect-Oriented Design. 10th European Software Engineering Conference Held Jointly with 13th ACM SIG-SOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), pages 116–175.

Sullivan, K. J., Griswold, W. G., Cai, Y., e Hallen, B. (2001). The Structure and Value of Modularity in Software Design. 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pag. 99–108. New York–USA.