Uma Abordagem para Recuperação da Arquitetura Dinâmica de Sistemas de Software

Hugo de Brito*, Henrique Rocha†, Ricardo Terra† e Marco Túlio Valente†

*Instituto de Informática
Pontifícia Universidade Católica de Minas Gerais
Email: hugobritobh@gmail.com

†Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Email: {hscr,terra,mtov}@dcc.ufmg.br

***Resumo*—Arquitetura de software se preocupa com a organização de mais alto nível de um sistema, incluindo seus componentes principais e os relacionamentos existentes entre eles. No entanto, apesar de sua inegável importância, diagramas com valor arquitetural não existem para a maioria dos sistemas ou, quando existem, eles não mais refletem a implementação atual desses sistemas. Técnicas de engenharia reversa podem ser então usadas então para extrair uma representação da arquitetura implementada de um sistema. No entanto, essas técnicas normalmente permitem a extração apenas de diagramas que revelam a arquitetura estática de um sistema, tais como diagramas de classe. Por outro lado, o estudo da arquitetura dinâmica de um sistema pode ser particularmente útil quando se deseja formar um primeiro entendimento do funcionamento de um sistema ou avaliar o impacto de uma possível manutenção no mesmo. Sendo assim, descreve-se neste artigo uma abordagem que permite a recuperação de um grafo de objetos a partir da execução de um sistema existente. O grafo proposto possui diversas características que – quando combinadas – o distinguem de outras abordagens semelhantes, incluindo: (a) suporte a grupos de objetos de maior granularidade, chamados domínios; (b) suporte aos diversos tipos de relacionamentos e entidades que podem existir em sistemas orientados por objetos; (b) suporte a sistemas *multi-thread*; (c) suporte a uma linguagem para definição de alertas associados a relacionamentos que são esperados (ou que não são esperados) em um sistema. Adicionalmente, descreve-se o projeto e a implementação de uma ferramenta para visualização dos grafos de objetos propostos. Por fim, apresenta-se um estudo de caso que mostra como essa ferramenta pode auxiliar na compreensão da arquitetura dinâmica de dois sistemas.**

I. Introdução

Arquitetura de software se preocupa em descrever os componentes principais de um sistema, bem como em especificar os relacionamentos possíveis entre esses componentes [13], [7], [21]. Assim, a definição da arquitetura de um sistema envolve um conjunto de decisões que são críticas para seu sucesso e que não poderão ser facilmente revertidas nas fases seguintes de seu desenvolvimento. No entanto, apesar de sua inegável importância, componentes e abstrações com valores arquiteturais não existem para a maioria dos sistemas ou, quando existem, eles não mais refletem a implementação atual desses sistemas [15], [9], [20].

Assim, técnicas de engenharia reversa são frequentemente usadas para extrair uma representação da arquitetura de um sistema [10], [27]. Normalmente, essas técnicas permitem a extração de diagramas que revelam a arquitetura estática de um sistema, tais como diagramas de classes [12], diagramas de pacotes [12] e matrizes de dependência estrutural [22]. Esses diagramas têm como principal vantagem o fato de serem recuperados diretamente do código fonte do sistema (isto é, sem a necessidade de execução). No entanto, eles apresentam uma versão parcial dos relacionamentos que são estabelecidos durante a execução de um sistema [2], [8], [14]. Por exemplo, diagramas estáticos não revelam relacionamentos estabelecidos por meio de polimorfismo e de chamada dinâmica de métodos, nem relacionamentos por meio do uso de reflexão computacional. Além disso, exatamente por serem estáticos, eles não possuem nenhuma informação sobre a ordem com que os diversos relacionamentos modelados foram de fato estabelecidos, o que é crucial para facilitar o entendimento por parte de usuários que não tenham familiaridade com os sistemas recuperados. Em outras palavras, para a compreensão de um sistema por meio de diagramas estáticos, desenvolvedores frequentemente têm a sensação de não saber por onde começar. Por fim, diagramas estáticos não fazem diferenças entre relacionamentos estabelecidos por diferentes *threads*, o que dificulta o entendimento de sistemas concorrentes (os quais têm se tornado cada vez mais comum em qualquer domínio de aplicação e não mais apenas na área de *software* complexo).

Por outro lado, existem técnicas de engenharia reversa que visam a extração de diagramas que revelam a arquitetura dinâmica de um sistema, tais como diagramas de objetos e de sequência [12]. Como principal vantagem, essas alternativas permitem expressar o fluxo de execução de um sistema, capturando inclusive relacionamentos decorrentes de chamadas dinâmicas e de reflexão computacional [23], [29]. No entanto, esses diagramas normalmente não são escaláveis, apresentando milhares de objetos mesmo para sistemas de pequeno porte [2], [3]. Além disso, modelos dinâmicos normalmente não fazem distinção entre objetos de mais baixo nível (exemplo:

`java.util.Date`) e objetos de maior valor arquitetural (exemplo: uma coleção de objetos do tipo `Customer`).

As soluções propostas para melhorar a escalabilidade de diagramas dinâmicos se baseiam em um mesmo princípio: agrupar objetos em unidades de maior granularidade (normalmente denominadas domínios [2], componentes [16], clusters [5] etc), de forma a permitir uma visão hierarquizada do diagrama. No nível mais alto dessa visão, são representados apenas grupos de objetos de maior valor arquitetural. Pode-se então expandir os grupos de uma visão de mais alto nível, de forma a fornecer mais detalhes sobre seus elementos. Esse processo pode ser repetido sucessivas vezes, até se chegar a um grafo de objetos totalmente plano, onde cada elemento representado corresponde a um objeto do programa base. Basicamente, existem duas propostas de soluções para agrupar objetos em unidades de maior granularidade: soluções automáticas (por exemplo, usando algoritmos de clusterização [6], [5]) e soluções manuais (por exemplo, por meio de anotações no código [2], [3]). Usualmente, soluções automáticas têm a desvantagem de não gerar grupos de objetos semelhantes àqueles que seriam definidos pelo arquiteto do sistema. Por outro lado, soluções via anotações são invasivas, requer um conhecimento prévio do sistema e uma inspeção detalhada no código para anotar cada classe com marcações indicando seu papel na arquitetura do sistema (exemplo: classes da camada de visão seriam anotadas com uma marcação `View`).

Neste artigo, descreve-se uma abordagem que permite a recuperação de um grafo de objetos a partir da execução de um sistema existente. O grafo proposto possui diversas características que o distinguem de outras abordagens semelhantes, incluindo: (a) suporte a grupos de objetos de maior granularidade, chamados domínios, os quais são definidos de forma não-invasiva, por meio de uma linguagem de expressões regulares; (b) suporte aos diversos tipos de relacionamentos e entidades que podem existir em um sistema orientado por objetos, incluindo relacionamentos devido a chamadas dinâmicas, reflexão computacional e relacionamentos entre objetos e campos estáticos de classes; (c) suporte a sistemas *multi-thread* por meio do uso de cores para diferenciar objetos criados por *threads* diferentes; (d) suporte a uma linguagem para definição de alertas associados a relacionamentos que são esperados (ou que não são esperados) em um sistema. Adicionalmente, descreve-se o projeto e a implementação de uma ferramenta para visualização dos grafos de objetos propostos. Essa ferramenta foi projetada de forma que ela pode ser "acoplada" a um sistema existente, permitindo assim a visualização do grafo de objetos proposto à medida que o programa base está sendo executado (ou seja, diferentemente de outras abordagens, não é preciso executar o sistema, para gerar um arquivo com o rastro da execução, que somente então será exibido para os usuários finais). Por fim, apresenta-se um estudo de caso que demonstra como a ferramenta pode auxiliar na compreensão da arquitetura dinâmica de dois sistemas.

O restante deste artigo está organizado conforme descrito a seguir. Na Seção II, descrevem-se o grafo de objetos proposto, incluindo uma descrição de seus principais elementos e alguns exemplos. Na Seção III, descreve-se a *Linguagem de Alertas*, usada para notificar os arquitetos sobre o estabelecimento de relacionamentos que são desejados (ou que não são desejados) em um sistema. Em seguida, a Seção IV descreve a ferramenta proposta para visualização dos grafos de objetos propostos, bem como de eventuais alertas definidos pelos arquitetos. A Seção V apresenta um estudo de caso, envolvendo o uso da ferramenta para visualizar e compreender dois sistemas construídos segundo o padrão arquitetural MVC (*Model-View-Controller*). A Seção VI discute trabalhos relacionados e a Seção VII conclui o artigo, resumindo as principais contribuições dos grafos de objetos propostos e delineando possíveis linhas de trabalhos futuros.

II. Grafo de Objetos

Um *OG* (*Object Graph*) é um grafo direcionado que representa o comportamento dinâmico dos objetos de um sistema. Os vértices de um *OG* representam todos os objetos e algumas das classes de um sistema. As arestas representam os possíveis relacionamentos entre os vértices do grafo. Detalhes sobre os vértices e arestas de um *OG* são fornecidos a seguir.

Vértices: Um *OG* admite dois tipos principais de vértices. Vértices na forma de um círculo são usados para representar objetos. Vértices na forma de um quadrado denotam classes. Vértices na forma de círculo têm o mesmo tempo de vida de objetos (isto é, são criados quando se instancia um objeto no programa e são destruídos quando o objeto que representam é alvo do coletor de lixo de Java). Vértices na forma de um quadrado são criados para representar classes com membros estáticos que referenciam ou fazem uso de serviços providos por objetos. Ou seja, nem todas as classes de um programa são representadas em um *OG*, mas apenas aquelas que possuem atributos estáticos que referenciam objetos ou que possuem métodos estáticos que acessam serviços providos por objetos.

O nome de um vértice – seja ele objeto ou classe – é uma estrutura composta por três campos. O primeiro campo é um inteiro não-negativo e sequencial que indica a ordem de criação dos vértices no grafo. Por convenção, o primeiro vértice criado – normalmente, representando a classe que contém o método `main` do sistema – recebe o número zero. O objetivo desse inteiro é viabilizar uma leitura "sequencial" do grafo, começando pelo ponto de entrada da aplicação, passando então para os objetos criados ou referenciados a partir dessa classe e assim por diante.

O segundo campo do nome de um vértice indica o nome da classe do objeto representado (se vértice circular) ou o nome da classe com membros estáticos que referenciam ou acessam serviços de objetos (se vértice quadrangular). Por fim, o terceiro campo do nome de um vértice representa a sua cor. Em um *OG*, usam-se cores para diferenciar vértices criados ou referenciados pelas *threads* de um sistema. Todos os vértices criados pela *thread* principal recebem a cor preta. Caso o programa crie outra *thread*, uma nova cor é automaticamente usada para representar os objetos criados

pela mesma. Em resumo, os vértices de um *OG* possuem múltiplas cores, as quais denotam as *threads* em que foram referenciadas.

Arestas: Em um *OG* arestas representam relacionamentos entre os objetos e classes do grafo. Suponha que o_1 e o_2 são vértices circulares (representando objetos) e que c_1 e c_2 são vértices quadrangulares (representando classes). Uma aresta direcionada (o_1,o_2) indica que o_1 – em algum momento do seu tempo de vida – adquiriu uma referência para o_2. Essa referência pode ter sido adquirida por meio de um campo, de uma variável local ou de um parâmetro formal de método. Já uma aresta direcionada (o_1,c_1) indica que o_1 – em algum momento do seu tempo de vida – chamou um método estático implementado por c_1.

Por outro lado, uma aresta direcionada (c_1,o_1) indica que c_1 – em algum momento da execução do programa – adquiriu uma referência para o_1. Essa referência foi adquirida por meio de um campo estático da classe c_1. Por fim, uma aresta direcionada (c_1,c_2) indica que c_1 – em algum momento da execução do programa – chamou um método estático implementado por c_2.

As arestas descritas anteriormente são inseridas no grafo que logo o relacionamento representado seja estabelecido durante a execução do programa. Quando um vértice é removido do grafo, todas as arestas que chegam e saem do mesmo são também removidas. Para aumentar a legibilidade do grafo, não são mostradas arestas representando auto-relacionamentos (isto é, arestas que saem e chegam no mesmo vértice).

Pacotes e Domínios: Como em qualquer representação visual da execução de um programa, o tamanho e a quantidade de informações representadas em um *OG* tendem a crescer rapidamente, podendo comprometer a legibilidade do grafo mesmo em aplicações pequenas. Para garantir que *OG* são escaláveis até sistemas de grande porte, dispõe-se de um recurso para sumarização de vértices. Basicamente, ao visualizar um *OG* pode-se optar por sumarizar alguns vértices em um vértice único, o qual é representado por meio de um hexágono. Conforme descrito a seguir, existem dois critérios para sumarização de vértices.

O primeiro critério consiste em representar apenas o pacote onde se encontram as classes dos objetos representados no grafo. Ou seja, todos os objetos que pertencem a classes de um mesmo pacote são agrupados em um vértice único. Suponha dois vértices representando os pacotes p_1 e p_2. Existirá uma aresta direcionada (p_1,p_2) se existir pelo menos um elemento de p_1 a um elemento de p_2 na versão não-sumarizada do *OG*.

O segundo critério de sumarização consiste em agrupar os vértices de acordo com conjuntos definidos pelos próprios usuários de um *OG*. O objetivo é fornecer um mecanismo mais flexível para sumarização de vértices do que aquele baseado exclusivamente na divisão estática de um sistema em pacotes. Basicamente, um domínio é uma coleção de objetos de classes especificadas pelo usuário, por meio da seguinte sintaxe:

```
domain name: classes
```

onde `name` é o nome do domínio e `classes` é uma lista de classes separadas por vírgulas. Para fins de sumarização, todos os objetos das classes listadas são considerados como pertencentes ao domínio `name`. Para facilitar a definição das classes de um domínio podem ser usadas expressões regulares ou o operador `+` (exemplo: `A+` denota a classe `A` e suas subclasses).

A. Exemplos

Para ilustrar os elementos básicos de um *OG* mostram-se nesta subseção alguns exemplos de grafo, extraídos de trechos de código hipotéticos.

Exemplo 1 (Vértices e Arestas): Suponha o código mostrado na Listagem 1. Neste código, a classe `Main` instancia um objeto do tipo `Invoice` e chama o método `load` do mesmo (linhas 4-5). A execução do método `load` cria um `ArrayList` e insere um `Product` no mesmo (linhas 12-14).

```
1  public class Main {
2      private static Invoice invoice;
3      public static void main(String[] args) {
4          invoice = new Invoice();
5          invoice.load();
6      }
7  }
8
9  public class Invoice {
10     private Collection<Product> col;
11     public void load(){
12         col = new ArrayList<Product>();
13         Product p = new Product();
14         col.add(p);
15     }
16 }
```

Listagem 1. Código do Exemplo 1

A Figura 1 apresenta o *OG* gerado pela execução do código mostrado na Listagem 1. Esse *OG* possui um vértice quadrangular (representando a classe que contém o `main` da aplicação) e três vértices circulares, representando os objetos criados durante a execução do programa.

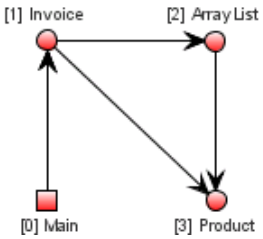

Figura 1. *OG* do Exemplo 1

O *OG* mostrado representa de modo compacto a execução do programa. Acompanhando o inteiro sequencial associado a cada vértice, pode-se verificar que a classe `Main` (vértice 0) acessou um objeto do tipo `Invoice` (vértice 1), que por sua vez acessou um objeto do tipo `ArrayList` (vértice 2).

Por fim, pode-se verificar que o último objeto criado foi do tipo `Product` (vértice 3). Em algum ponto da execução do programa, esse objeto foi acessado pelos objetos `Invoice` (responsável pela sua criação) e `ArrayList` (responsável pelo seu armazenamento).

Exemplo 2 (Threads): Suponha o código mostrado na Listagem 2. Nesse código, a classe `Main` instancia e ativa duas *threads* do tipo `Box` (linhas 3-4). As *threads* criadas simplesmente instanciam um objeto do tipo `Product` (linha 10).

```
1  public class Main{
2     public static void main(String[] args) {
3        new Box().start();
4        new Box().start();
5     }
6  }
7
8  public class Box extends Thread {
9     public void run() {
10       new Product();
11    }
12 }
```

Listagem 2. Código do Exemplo 2

A Figura 2 apresenta o *OG* gerado pela execução do código da Listagem 2. Nesse *OG*, pode-se verificar que a classe `Main` (vértice 0) acessou dois objetos do tipo `Box` (vértices 1-2). Pode-se verificar também que cada objeto `Box` acessou um objeto do tipo `Product` (vértices 3-4). Mais importante, no *OG* mostrado, os vértices que representam `Product` possuem cores diferentes, visto que eles foram criados por *threads* diferentes.

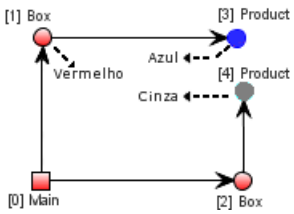

Figura 2. *OG* do Exemplo 2

Exemplo 3 (Domínios): Suponha um sistema hipotético, construído na arquitetura MVC (*Model-View-Controller*). Para prover uma visão arquitetural (isto é, de alto nível) e dinâmica (isto é, que expresse os relacionamentos entre os objetos desse sistema), suponha que foram definidos os seguintes domínios:

```
1 domain View:  myapp.view.IView+
2 domain Controller: myapp.controller.*
3 domain Model: "myapp.model.[a-zA-Z0-9/.]*DAO"
4 domain Swing: javax.swing.**
5 domain Hibernate: org.hibernate.**
```

Listagem 3. Exemplo de Definição dos Domínios

Nessa definição, o domínio `View` denota quaisquer objetos da classe `myapp.view.IView` e de suas subclasses. O domínio `Controller` denota objetos de quaisquer classes do pacote `myapp.controller`. Já o domínio `Model` denota objetos de classes cujo nome inicia-se com `myapp.model` e termina com `DAO`. O operador ** seleciona todas as classes de pacotes com o prefixo especificado. Portanto, os domínios `Swing` e `Hibernate` denotam, respectivamente, objetos das classes do pacote `javax.swing` e `org.hibernate`, assim como objetos de quaisquer classes de pacotes internos aos pacotes relacionados.

A Figura 3 apresenta um possível *OG* gerado para o sistema MVC hipotético considerado nesse exemplo. Primeiro, veja que cinco vértices são em forma de hexágono, representando cada um dos domínios definidos anteriormente. Existe ainda um único vértice em forma de círculo, representando um objeto do tipo `Util`, o qual não pertence a nenhum dos domínios especificados. Ou seja, objetos pertencentes a um dos domínios definidos antes da geração do grafo são automaticamente sumarizados em um vértice em forma de hexágono; objetos que não são capturados por nenhum dos domínios definidos continuam sendo representados por meio de vértices circulares (no caso de objetos) ou quadrangulares (no caso de classes).

No *OG* da Figura 3, é possível observar que a divisão de domínios do sistema segue o padrão MVC. Por exemplo, existe uma comunicação em bidirecional entres os domínios `View` e `Controller` e entre os domínios `Controller` e `Model`. Ou seja, o *OG* mostrado revela com clareza que o domínio `Controller` desempenha o papel de um mediador entre os domínios `View` e `Model`, conforme previsto em arquiteturas MVC. Pode-se observar ainda que somente o domínio `View` está acoplado ao *framework* Swing e que somente o domínio `Model` está acoplado ao *framework* Hibernate.

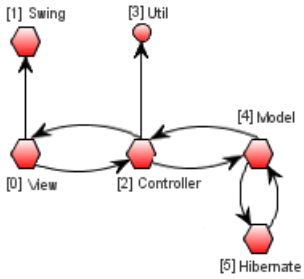

Figura 3. *OG* do Exemplo 3

III. Linguagem de Alertas

Grafos de objetos foram projetados para capturar de forma não-invasiva a arquitetura dinâmica de sistemas. Sendo assim, são ferramentas úteis a um arquiteto que esteja interessado em compreender o comportamento dinâmico dos sistemas sob sua responsabilidade. A fim de auxiliar tais arquitetos, foi definida uma linguagem para exibição de alertas em um *OG*. A ideia básica é associar alertas a relacionamentos que são esperados (ou que não são esperados) em um sistema. Quando

tais relacionamentos forem estabelecidos no *OG*, um alerta é exibido para o arquiteto.

Suponha um sistema onde o acesso a dados é realizado por meio de objetos que implementam o padrão DAO (*Data Access Objects*) [11]. Um arquiteto interessado em estudar e entender melhor os módulos de persistência desse sistema pode, por exemplo, definir um alerta que gere uma mensagem toda vez que um determinado domínio usar os serviços de objetos `DAO`. Como um segundo exemplo, dessa vez de relacionamento que não deveria existir, um arquiteto poderia definir um alerta para gerar uma mensagem toda vez que uma classe utilitária usar serviços de outros módulos do sistema. Esse relacionamento não deveria ocorrer no sistema em questão, visto que classes utilitárias não devem estabelecer dependências com sistemas clientes, de forma a preservar o potencial de reúso das mesmas.

Os alertas definidos pelos usuários de um *OG* são exibidos de duas formas: (a) mudando a cor das arestas relativas aos relacionamentos responsáveis pelo alerta; (b) gerando uma mensagem em uma janela de alertas, onde são informados detalhes sobre os alertas gerados durante a execução do sistema.

Sintaxe: Alertas são definidos por meio da seguinte gramática:

```
<alert_clause> ::= alert <domain> {, <domain>}
                   <relation> <domain> {, <domain>}

<domain> ::= [!] <string> | *
<relation> ::= access | create | depend
```

Nessa gramática, símbolos não-terminais são escritos entre < e > (tal como em `<domain>`). Colchetes são usados para delimitar símbolos opcionais (tal como em `[!]`, indicando que o terminal `!` é opcional). As chaves { e } (tal como em {`, <domain>`}) indicam que o elemento pode ter zero ou mais repetições. Símbolos terminais são escritos sem nenhum delimitador especial (tal como em `alert`, `access`, etc). O não-terminal `<string>` denota uma *string* (isto é, uma sequência de caracteres, sem aspas delimitadoras).

De acordo com essa gramática, uma cláusula de alerta define um relacionamento entre dois domínios. Isso é, o alerta será ativado quando for detectado um relacionamento do tipo especificado entre os domínios definidos na cláusula. Na definição de um domínio, pode-se usar o operador de negação `!` para denotar todos os objetos não pertencentes ao domínio especificado. Por exemplo, `!A` é um domínio que contém qualquer objeto não pertencente ao domínio `A`. Por fim, o símbolo `*` denota qualquer objeto, independentemente de seu domínio.

Para exemplificação da sintaxe de regras de alertas, suponha os objetos `a1` do domínio `A`, `b1` do domínio `B` e `c1` do domínio `C`, as seguintes regras funcionariam como a seguir:

- *alert A x*[1] *B*: Essa restrição destaca dependências do tipo `x` dos objetos do domínio `A` com objetos do domínio `B`. Logo, é destacada a aresta entre `a1` e `b1` se, e somente se, `a1` estabelece uma dependência do tipo `x` com `b1`.

- *alert A x !B*: Essa restrição destaca dependências do tipo `x` dos objetos do domínio `A` com objetos fora do domínio `B`. Logo, é destacada a aresta entre `a1` e `c1` se, e somente se, `a1` estabelece uma dependência do tipo `x` com `c1`.

- *alert !A x B*: Essa restrição destaca dependências do tipo `x` dos objetos fora do domínio `A` com objetos do domínio `B`. Logo, é destacada a aresta entre `c1` e `b1` se, e somente se, `c1` estabelece uma dependência do tipo `x` com `b1`.

Na definição de uma cláusula de alerta, podem ser especificados três tipos de relacionamentos entre os domínios que se deseja monitorar:

- `depend`: qualquer tipo de relacionamento, dentre aqueles que podem ser representados em um grafo de objetos. Em outras palavras, um alerta desse tipo será ativado quando durante a execução do programa, um objeto do tipo de origem referenciou algum objeto do domínio de destino. Essa referência pode ter sido armazenada em uma variável local, em um parâmetro formal ou em um campo do objeto do domínio de origem.

- `access`: um alerta desse tipo será ativado quando durante a execução do programa, um objeto do tipo de origem não apenas referencia um objeto do domínio de destino, como também acessa um campo ou chama um método desse objeto. Ou seja, `access` é um caso específico de um relacionamento do tipo `depend`. Por exemplo, um objeto pode referenciar um objeto de outro domínio (`depend`), mas não usar nenhum dos serviços providos por ele (`access`). Um exemplo são objetos de fachada, que simplesmente repassam referências recebidas como parâmetro para métodos escondidos por trás da fachada.

- `create`: indica que, durante a execução do programa, um objeto do domínio de origem criou um objeto do domínio de destino. Mais especificamente, essa criação deve ter ocorrido durante a execução de um método de um objeto do domínio de origem.

A. Exemplo

A Listagem 4 demonstra três exemplos de alertas (usando os domínios definidos anteriormente na Listagem 3). Nessa listagem, são definidos alertas para quando um objeto qualquer acessar um objeto do domínio `Hibernate` (linha 1). Define-se também um alerta para quando um objeto da classe `myapp.Util` acessar um objeto que não seja dessa mesma classe (linha 2). Com esse segundo alerta deseja-se verificar se classes utilitárias são auto-contidas, funcionando meramente

[1]O literal `x` se refere ao tipo da dependência, que pode ser mais abrangente (*depend*) ou mais específico (*access* e *create*).

como provedoras de serviços para módulos externos. Por fim, define-se um alerta destinado a monitorar se objetos do tipo `DAOImpl` são criados apenas pela classe fábrica dos mesmos (linha 3). Assim, quando um objeto que não seja do tipo `DAOFactory` tentar instanciar uma implementação de DAO (`DAOImpl`), um alerta será ativado.

```
1 alert * access Hibernate
2 alert myapp.Util access !myapp.Util
3 alert !DAOFactory create DAOImpl
```

Listagem 4. Exemplo da *Linguagem de Alerta*

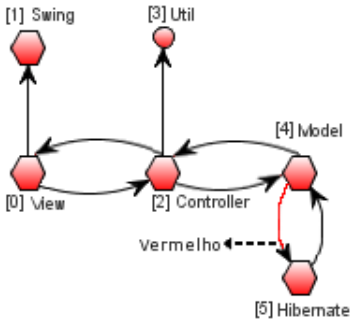

Figura 4. Verificação arquitetural de relacionamentos detectados pela *Linguagem de Alertas*

Pode-se notar que o *OG* destaca em vermelho a aresta que representa uma dependência entre o domínio `Model` e o domínio `Hibernate`, conforme especificado na definição do primeiro alerta da Listagem 4. Adicionalmente, em uma janela própria, esse alerta é detalhado, com mais informações sobre o objeto de origem e de destino do acesso responsável pela sua ativação.

Os relacionamentos especificados pelos dois últimos alertas da Listagem 4 não foram detectados em nenhum ponto da execução do programa. Assim, pode-se concluir que – pelo menos nessa execução – não foram detectadas dependências entre classes utilitárias e outras classes do programa; também não ocorreram instanciações de DAO fora das fábricas especificamente implementadas com tal finalidade.

IV. Ferramenta

A ferramenta *OG* (*Object Graph*) é constituída por três módulos: *OGT* (*Object Graph Tracer*), *OGV* (*Object Graph Visualization*) e a *Linguagem de Alertas*. Nas subseções seguintes são descritos em detalhes cada um desses módulos.

A. OGT

O módulo *OGT* é o responsável pelo rastro de execução e trabalha sobre objetos, métodos, atributos, coleções e *threads*. Para isso, utiliza a tecnologia *AspectJ* [19] – uma extensão orientada a aspectos para a linguagem Java que permite a injeção de código em uma aplicação de uma forma não intrusiva [1]. No *OGT*, existe uma classe de configuração na qual são definidos diversos parâmetros, por exemplo, exclusão do rastro de execução de determinadas classes, exibição de classes anônimas, visualização por pacotes ou utilização de domínios. Esses parâmetros auxiliam o arquiteto a refinar grafo à medida que vai se conhecendo o sistema ou uma determinada funcionalidade. É importante mencionar que o *OGT* é o módulo principal cujas informações são repassadas para os módulos *OGV* e para a *Linguagem de Alertas*.

B. OGV

O módulo *OGV* tem como objetivo a exibição e configuração do *OG*. Para a exibição do grafo foi utilizado uma biblioteca de software – denominada *JunG*[2] – que provê uma forma simples de visualização de dados. O *JunG* possui leiautes pré-definidos que ajudam na organização do *OG*, mas também permite que o arquiteto organize o grafo do modo como desejar. Além disso, pode-se escolher quais arestas serão detalhadas e ainda permite iniciar ou parar a captura do rastro de execução a qualquer momento.

C. Linguagem de Alertas

O módulo de *Linguagem de Alertas* (também chamado de *OGAL* (*Object Graph Alert Language*) é responsável pelo gerenciamento dos alertas arquiteturais. Este módulo verifica as regras de alertas definidas pelo arquiteto e monitora o sistema através das informações recebidas do *OGT*. Caso encontre alguma dependência a alertar – a partir das regras definidas pelo arquiteto – ele envia uma mensagem para que o *OGV* destaque tal dependência. A Figura 5 exibe o diagrama arquitetural do módulo *OGAL*, destacando suas entradas e saídas.

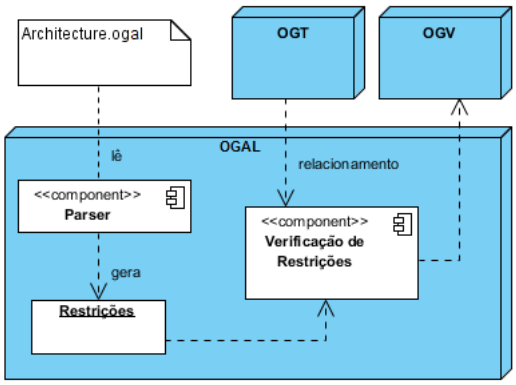

Figura 5. Diagrama arquitetural da *Linguagem de Alertas*

Basicamente, esse módulo é dividido nos seguintes componentes:

- *Parser*: esse componente efetua a leitura do arquivo `architecture.ogal` – que contém as definições dos domínios e dos alertas – e cria uma estrutura de dados responsável por armazenar as regras de alertas que são verificadas *a posteori*;
- *Verificação das Restrições*: continuamente, esse componente recebe os rastros de execução gerados pelo *OGT* e cada dependência é conferida com as regras armazenadas

[2]http://jung.sourceforge.net/

pelo *Parser*. Caso a dependência viole alguma regra, tal fato é informado ao *OGV* que destacará o alerta arquitetural (na cor vermelha).

É importante salientar que, como se trata de análise em tempo de execução, não é possível mapear todas dependências do sistema *a priori*. Assim, cada dependência é verificada no momento em que é estabelecida.

V. Cenários de Aplicação

Para ilustrar a aplicação da ferramenta *OG* tanto no quesito de visualização quanto no uso da *Linguagem de Alertas*, nesta seção são apresentados alguns cenários de aplicação. O objetivo central é demonstrar os benefícios que o conceito de *OG* pode proporcionar a arquitetos de software.

A. Sistemas Alvos

Para esta seção, são utilizados dois sistemas. O sistema *myAppointments* é um sistema simples de gerenciamento de informações pessoais. De forma sucinta, esse sistema segue o padrão arquitetural MVC e possui 1.215 *LOC*, 16 classes e três interfaces. A Figura 6 ilustra a tela principal do sistema. Basicamente, *myAppointments* permite aos usuários criar, pesquisar, atualizar e remover compromissos. Esse sistema foi originalmente descrito para ilustrar o emprego de soluções para conformação estática de arquiteturas de software [20].

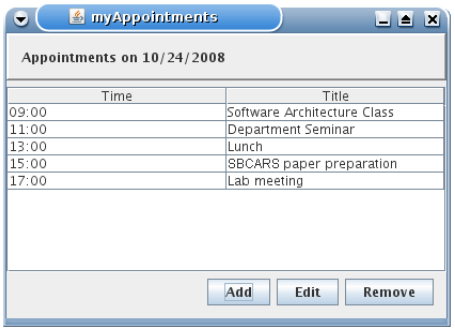

Figura 6. Sistema *myAppointments*

O segundo sistema, *JHotDraw*, é um *framework* bem conhecido que auxilia no desenvolvimento de aplicativos gráficos. Esse *framework* possui 15 *KLOC* e 200 classes. Do ponto de vista arquitetural, faz o uso de diversos padrões de projeto e segue também o padrão arquitetural MVC.

B. Cenários de Aplicação

O objetivo principal da visualização de um sistema é facilitar o entendimento do comportamento do sistema ou de uma funcionalidade específica. É importante mencionar que, conforme abordado na Seção II, existem diversos filtros, recursos e visões que permitem modificar o grafo de forma que facilite o entendimento por parte do arquiteto de software.

1) Manutenção myAppointments: Dado o sistema *myAppointments*, suponha que o arquiteto esteja planejando uma modificação na remoção de um compromisso. Contudo, ele desconhece o comportamento do sistema no processo de remoção de registros. Para isso, o arquiteto pode utilizar a ferramenta *OG* para capturar o rastro de execução do processo de remoção. Por exemplo, em uma primeira visualização, pode-se optar pela visualização em pacotes. Ainda, caso deseje mais detalhes, pode-se optar pela visualização de objetos.

Concretizando o exemplo acima, na Figura 7 é apresentada o grafo de pacotes somente do processo de remoção de um compromisso. Como pode se observar, foram criados cinco vértices (representando pacotes): `myapp.controller` (controle), `myapp.view` (visão), `myapp.model` (modelo), `org.hsqldb.jdbc` (persistência) e `myapp.model.domain` (objetos de domínio). As arestas representam as comunicações e, nesse caso, percebe-se que o pacote de controle estabelece comunicação com pacotes de visão e modelo, e que o modelo se comunica com o módulo de persistência e com os objetos de domínio.

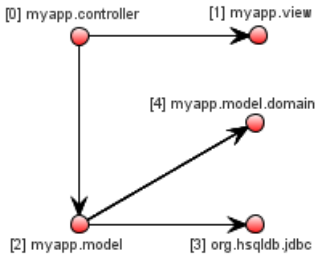

Figura 7. *OG* com sumarização por pacotes

No entanto, o arquiteto pode estar interessado em uma visualização mais detalhada do sistema. Para isso, o mesmo grafo representado na Figura 7 pode ser visualizado em granularidade mais fina – nível de objetos – como representado na Figura 8. Sabe-se que um *OG* em granularidade fina não é escalável. No entanto, um grafo detalhado de apenas uma determinada funcionalidade é mais viável. Como pode se observar, foram agora criados sete vértices (ao invés de cinco, no grafo de pacotes): objeto `AgendaController` (ponto de entrada da aplicação), objetos `AgendaView` e `AgendaDAO`, classe estática `DB` e objetos `JDBCConnection`, `DAOCommand` e `App`. Veja que esse grafo oferece mais informações sobre o comportamento do sistema, mostrando, por exemplo, que objetos DAO são usados para acesso a dados, que a comunicação com o banco de dados é feita por meio de JDBC etc.

O arquiteto pode ainda definir domínios para melhor representar o papel arquitetural de grupos de objetos. Por exemplo, suponha que o arquiteto defina o domínio `Model` como sendo composto por objetos de quaisquer classes do pacote `myapp.model`, conforme descrito na Listagem 5.

```
1 domain Model: myapp.model.**
```

Listagem 5. Definição de domínio no `myAppointments`

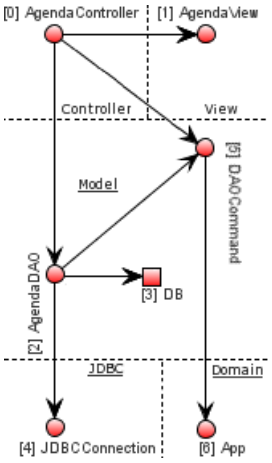

Figura 8. *OG* em granularidade fina

A partir da definição desse domínio, o mesmo grafo das Figuras 7 e 8 é visualizado na Figura 9. Nesse grafo, todos os vértices que representavam objetos ou classes de modelo – objetos `AgendaDAO`, `DAOCommand`, `App` e a classe estática `DB` – foram sumarizados em um único vértice, `Model`. Assim, foram agora criados somente quatro vértices, o que simplifica e facilita o entendimento do grafo pelo arquiteto.

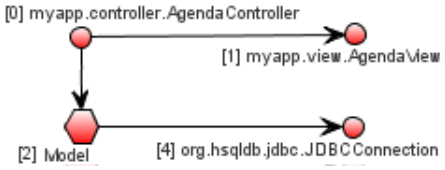

Figura 9. *OG* com definição do domínio `Model`

Em resumo, quando do interesse do arquiteto, a abordagem permite recuperar diagramas mais detalhados que aqueles sumarizados por pacotes (como o diagrama mostrado na Figura 8). Por outro lado, se necessário, ela é flexível a ponto de também permitir a recuperação de diagramas de mais alto nível que aqueles sumarizados por pacotes (como o diagrama mostrado na Figura 9).

Em todos os exemplos analisados neste cenário de aplicação, foram filtrados os objetos *Java AWT*, da biblioteca padrão de Java (*java.lang*) e utilitários do sistema, uma vez que eles não são relevantes para o entendimento do sistema.

2) Arquitetura Macro do JHotDraw: Para entendimento da arquitetura do segundo sistema usado como estudo de caso neste artigo, foi primeiro gerado o grafo a partir da inicialização do sistema *JHotDraw* sem qualquer tipo de filtro ou recurso de sumarização. Dessa maneira, foi gerado um grafo com quase mil vértices. Conforme pode ser observado na Figura 10, esse grafo é ilegível.

Para tornar o grafo mostrado legível e útil, ele foi novamente capturado com uso de domínios. A definição de domínios tomou como base a divisão de classes feita por Abi-Antoun e Aldrich [3]. Nessa definição, que segue o padrão

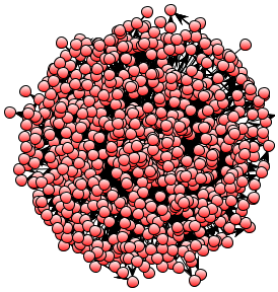

Figura 10. *OG* do *JHotDraw* sem filtro ou sumarização

arquitetural MVC, objetos de apresentação, como instâncias de `DrawingEditor` e `DrawingView`, foram definidos em domínios com prefixo `View`. Os objetos responsáveis pela lógica de apresentação, como instâncias de `Tool`, `Command` e `Undoable`, foram definidos em domínios com prefixo `Ctrl`. Por fim, objetos de modelo, como instâncias de `Drawing`, `Figure` e `Handle`, foram definidos em domínios com prefixo `Model`.

Assim, foram definidos dez domínios – três relacionados à visão, quatro ao controle e três ao modelo – conforme pode ser observado na Figura 11. A fim de deixar o grafo ainda mais legível, foi utilizado um recurso da ferramenta em que vértices que se comunicam em ambos os sentidos têm suas arestas unificadas em uma única aresta bidirecional. Ao contrário da Figura 10, a figura em questão traz diversas informações ao arquiteto. Inicialmente, é possível observar que não houve violação nas camadas do padrão MVC.

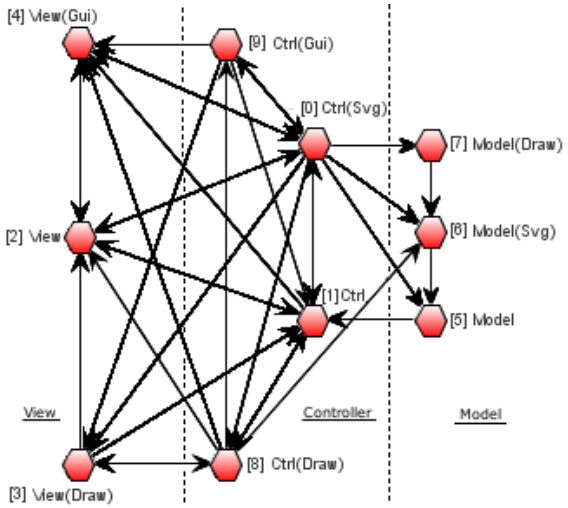

Figura 11. *OG* do *JHotDraw* com definição de domínios

Ademais, esse cenário indica que, para a tarefa de compreensão de um sistema, é importante que se obtenha inicialmente uma visão macro (de granularidade grossa) do sistema e, na medida em que se compreende a arquitetura, se detalha a granularidade do grafo. Por exemplo, caso o arquiteto deseje mais detalhes de um certo domínio, ele pode explodi-lo, isto é, desagrupar o domínio de forma que

seja exibida toda a comunicação interna entre seus objetos e classes.

3) Linguagem de Alertas: Quando se faz o uso da API de reflexão de Java, técnicas de análise dinâmica conseguem detectar violações que não são detectadas por meio de análise estática. Com análise estática não é possível obter a informação de qual objeto foi criado dinamicamente e, nem mesmo, de quais atributos ou métodos de um objeto foram acessados por reflexão. Esse é um ponto em que a análise dinâmica se sobressai em relação à análise estática.

Um cenário comum é o uso de reflexão para registrar um *driver* de um SGBD em uma aplicação Java. Isso é realizado por meio da instanciação de uma classe específica do SGBD, denominada *driver*. Normalmente, o nome qualificado dessa classe é armazenado em um arquivo texto ou mesmo inserido diretamente no código fonte, conforme ilustrado na Linha 1 da Listagem 6. Nesse exemplo, o *driver* do HSQLDB é registrado na aplicação por meio de reflexão.

```
1 Class.forName("org.hsqldb.JdbcDriver");
2 ...
3 Connection conn = DB.getConnection(...);
```

Listagem 6. Pseudo-código registrando o *driver* do HSQLDB

Suponha que somente o HSQLBD seja suportado pelo sistema. Logo, pode-se utilizar a ferramenta *OG* – completamente baseada em análise dinâmica – para garantir que, se o *driver* de conexão ao SGBD for alterado, um alerta será exibido ao arquiteto avisando sobre essa mudança. Isso é de extrema importância, uma vez que a mudança do SGBD sem uma prévia inspeção de um arquiteto, pode resultar em diversos problemas na aplicação. Por exemplo, instruções SQL que possuem códigos específicos do HSQLDB deixarão de funcionar.

Isso é facilmente realizado por meio das definições apresentadas na Listagem 7, que alerta se a classe `DB` – responsável pela conexão ao SGBD – criar qualquer objeto que não seja o *driver* do HSQLDB ou fora do domínio da API de Java SQL. Em outras palavras, qualquer tentativa de acessar outro SGBD é exibida no grafo de visualização como um alerta arquitetural.

```
1 domain JavaSql: java.sql.**
2 domain DB:      myapp.model.DB
3 alert DB create !org.hsqldb.JdbcDriver,
4                 !JavaSql
```

Listagem 7. Definições de alertas caso haja mudança do SGBD

VI. Trabalhos Relacionados

Scholia é uma abordagem para recuperação da arquitetura dinâmica (*run-time architecture*) que apresenta duas diferenças principais em relação à solução descrita neste artigo [2], [3]. Primeiro, a abordagem tenta inferir relacionamentos dinâmicos por meio de análise estática. As vantagens neste caso são que Scholia consegue capturar relacionamentos presentes em qualquer execução de um sistema (e não apenas aqueles presentes em uma execução particular, como a abordagem proposta neste artigo). No entanto, Scholia não consegue capturar, por exemplo, informações sobre a cardinalidade dos relacionamentos. Por exemplo, consegue-se determinar que uma coleção é composta por elementos do tipo `A`, mas não é possível saber quantos objetos existem nessa coleção. Como uma segunda diferença principal, Scholia requer o uso de anotações no código, para informar em quais domínios objetos de uma classe deverão ser sumarizados, para fins de visualização. Essa exigência pode inviabilizar o uso de Scholia em sistemas existentes e mesmo em novos sistemas. Via de regra, arquitetos, desenvolvedores e mantenedores são reticentes à inserção de anotações no código, o que além de ser uma tarefa custosa, implica em um acoplamento do sistema a uma ferramenta específica. ArchJava é uma linguagem para definição de arquiteturas que sofre de problemas até maiores nesse sentido, pois requer o aprendizado de uma extensão de Java com suporte sintático a abstrações arquiteturais, tais como componentes e conectores [4].

Womble é outra solução para recuperação de diagramas de objetos por meio de técnicas de análise estática [14]. Portanto, apresenta as mesmas vantagens e desvantagens de Scholia, no tocante à precisão dos relacionamentos recuperados pela ferramenta. No entanto, diferente de Scholia, Womble não disponibiliza recursos para sumarização de objetos em unidades de maior granularidade. Assim, grafos de objetos recuperados por Womble possuem milhares de objetos, mesmo para sistemas pequenos.

Discotect é uma ferramenta destinada a recuperar a arquitetura dinâmica de um sistema [29], [23]. No entanto, em vez de diagramas de objetos, Discotect extrai uma visão arquitetural baseada em conectores e componentes. Para isso, Discotect instrumenta o sistema alvo, de forma a permitir a captura de eventos que ocorrem durante a sua execução (por exemplo, chamadas de métodos). O sistema requer que arquitetos definam um mapeamento entre os eventos monitorados e componentes/conectores. Portanto, esse mapeamento requer um conhecimento mais detalhado do sistema alvo que aquele demandado para definição de domínios em *OG*.

Briand, Labiche, Leduc descrevem uma abordagem para extração de diagramas de sequência a partir da execução de um sistema alvo. De forma similar à ferramenta descrita neste trabalho, a instrumentação do sistema base é feita por meio de orientação por aspectos. No entanto, a instrumentação proposta gera eventos que são armazenados em um repositório. A ferramenta de visualização proposta gera os diagramas de sequência de forma *off-line*, lendo os eventos previamente salvos nesse repositório. A abordagem proposta não permite a definição de unidades de maior granularidade do que objetos. Por fim, a abordagem proposta tem como vantagem o fato de permitir a recuperação de diagramas de sequência de sistemas distribuídos, implementados em Java RMI [28].

A linguagem para definição de alertas em um *OG* foi inspirada na linguagem para conformação arquitetural DCL

(*Dependency Constraint Language*) [24], [26], [25]. Basicamente, DCL permite definir dependências aceitáveis e inaceitáveis de acordo com a arquitetura planejada de um sistema. Uma vez definidas, tais restrições são verificadas por uma ferramenta de conformação integrada à plataforma Eclipse. Portanto, DCL é uma linguagem voltada para conformação arquitetural, usando para isso análise estática. Já a linguagem de alertas proposta neste artigo é voltada para ajudar arquitetos a detectar relações importantes em um grafo de objetos. Por relações importantes, entendemos tanto relações esperadas no sistema, como relações que representam violações arquiteturais. Em resumo, a linguagem de alertas proposta no trabalho é um instrumento extra, do qual pode dispor um arquiteto interessado em entender melhor detalhes de um sistema.

Também com intuito de prover conformação arquitetural, Modelos de Reflexão são diagramas que detectam ausências, convergências e divergências entre a arquitetura planejada e a arquitetura concreta de um sistema [17], [18]. No entanto, modelos de reflexão pressupõem que os arquitetos já tenham uma visão de alto nível da arquitetura do sistema alvo.

VII. Conclusões

Neste trabalho, foi apresentado uma abordagem para recuperação de grafos de objetos a partir da execução de um sistema orientado por objetos. As principais contribuições dos grafos propostos são as seguintes: (a) suporte a grupos de objetos de maior granularidade; (b) suporte a diversos tipos de relacionamentos que podem existir em sistemas orientados por objetos; (c) suporte a sistemas *multi-thread*; (d) suporte a uma linguagem para definição de alertas associados a relacionados esperados (ou que não são esperados) em um sistema. Implementou-se também uma ferramenta para visualização dos grafos de objetos propostos, a qual permite a visualização *online* do grafo de objetos de um sistema, à medida que esse é executado. Por fim, com apoio dessa ferramenta, os grafos de objetos propostos foram usados para visualizar a arquitetura de dois sistemas.

Como trabalho futuro, pretende-se realizar novos estudos de caso, que melhor demonstrem os benefícios (e as limitações) da abordagem proposta neste artigo. Pretende-se também integrar a ferramenta proposta ao ambiente de desenvolvimento Eclipse.

Agradecimentos

Este trabalho foi apoiado pela FAPEMIG, CAPES e CNPq.

Referências

[1] C. W. A. Vasconcelos, R. Cêpeda. An approach to program comprehension through reverse engineering of complementary software views. In *PCODA: 1st International Workshop on Program Comprehension through Dynamic Analysis*, pages 58–62, Pittsburgh, USA, 2005.

[2] M. Abi-Antoun and J. Aldrich. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In *24th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 321–340, 2009.

[3] M. Abi-Antoun and J. Aldrich. Static extraction of sound hierarchical runtime object graphs. In *TLDI 09: Proceedings of the 4th international workshop on Types in language design and implementation*, pages 51–64, New York, NY, USA, 2009. ACM.

[4] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *22nd International Conference on Software Engineering (ICSE)*, pages 187–197, 2002.

[5] N. Anquetil and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *5th Working Conference on Reverse Engineering (WCRE)*, pages 235–255, 1999.

[6] N. Anquetil and T. C. Lethbridge. Ten years later, experiments with clustering as a software remodularization method. In *16th Working Conference on Reverse Engineering (WCRE)*, page 7, 2009.

[7] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd, edition, 2003.

[8] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed java software. *IEEE Transacions on Software Engineering*, 32(9):642–663, 2006.

[9] P. Clements and M. Shaw. The golden age of software architecture revisited. *IEEE Software*, 26(4):70–72, 2009.

[10] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[11] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[12] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 2003.

[13] D. Garlan and M. Shaw. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[14] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. *IEEE Transactions on Software Engineering*, 27(2):156–169, 2001.

[15] J. Knodel, D. Muthig, M. Naab, and M. Lindvall. Static evaluation of software architectures. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–294, 2006.

[16] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[17] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *3rd Symposium on Foundations of Software Engineering (FSE)*, pages 18–28, 1995.

[18] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[19] E. org. Aspectj. http://eclipse.org/aspectj/, 2004.

[20] L. Passos, R. Terra, R. Diniz, M. T. Valente, and N. das Chagas Mendonca. Static architecture conformance checking – an illustrative overview. *IEEE Software*, 2010. To appear.

[21] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, 1992.

[22] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 167–176, 2005.

[23] B. R. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, 2006.

[24] R. Terra and M. T. Valente. Towards a dependency constraint language to manage software architectures. In *Second European Conference on Software Architecture (ECSA)*, volume 5292 of *Lecture Notes in Computer Science*, pages 256–263. Springer, 2008.

[25] R. Terra and M. T. Valente. Verificação estática de arquiteturas de software utilizando restrições de dependência. In *II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*, pages 1–14, 2008.

[26] R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094, 2009.

[27] P. Tonella. Reverse engineering of object oriented code (tutorial). In *27th International Conference on Software Engineering (ICSE)*, pages 724–725, 2005.

[28] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232, 1996.

[29] H. Yan, D. Garlan, B. R. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A system for discovering architectures from running systems. In *26th International Conference on Software Engineering (ICSE)*, pages 470–479, 2004.