ArchLint: Uma Ferramenta para Detecção de Violações Arquiteturais usando Histórico de Versões

Cristiano Maffort[1], Marco Tulio Valente[1], Mariza A. S. Bigonha[1], Leonardo H. Silva[1], Gladston Junio Aparecido[2]

[1]Departamento de Ciência da Computação, UFMG
[2]Departamento de Ciência da Computação, Faculdade Pitágoras

{maffort,mtov,mariza,leonardosilva}@dcc.ufmg.br
gladston.aparecido@pitagoras.com.br

Resumo. *Erosão arquitetural é um problema que ocorre, recorrentemente, durante a evolução de sistemas de software. Neste artigo, apresenta-se ArchLint, uma ferramenta para conformidade arquitetural que combina análise estática e histórica de código fonte. ArchLint utiliza quatro heurísticas para detectar ausências e divergências. ArchLint foi aplicada em cinco sistemas e, como resultado, detectou 313 violações arquiteturais, com uma precisão de 62%.*

Abstract. *Architectural erosion is a problem that frequently occurs during the evolution of software systems. In this article, we present ArchLint, a tool for architecture conformance that based on a combination of static and historical source code analysis. ArchLint relies on four heuristics for detecting both absences and divergences. ArchLint was applied in five systems and as a result, the tool has detected 313 architectural violations with an precision of 62%.*

1. Introdução

Conformidade arquitetural é uma atividade chave para controle da qualidade em produtos de software. Basicamente, seu propósito é revelar lacunas entre a arquitetura concreta (implementada) e a arquitetura planejada de um software [Passos et al. 2010]. As principais técnicas para conformidade arquitetural utilizam modelos de reflexão [Murphy et al. 2001] ou linguagens de domínio específico, como DCL [Terra and Valente 2009]. Entretanto, em ambos os casos, é necessário definir manualmente uma representação da arquitetura planejada do sistema, incluindo seus componentes de alto nível, bem como as relações de dependência entre esses componentes. Esta arquitetura planejada, ou modelo de restrições arquiteturais, é então comparada com o código fonte do sistema, com objetivo de detectar eventuais violações das restrições estabelecidas.

As atuais técnicas de conformidade arquitetural, tipicamente, agrupam as violações arquiteturais em duas categorias: *ausências* e *divergências* [Passos et al. 2010]. Uma ausência ocorre quando uma dependência definida pela arquitetura planejada não está presente no código fonte. Por outro lado, divergências ocorrem quando dependências existentes no código fonte não estão em conformidade com a arquitetura planejada.

Na prática, a introdução de anomalias de codificação, que não são aderentes à arquitetura planejada, é um problema bastante comum [Knodel and Popescu 2007], capaz de tornar mais difícil as tarefas de manutenção de um sistema. Por outro lado, a

aplicação das atuais técnicas de conformidade arquitetural requerem um esforço considerável [Passos et al. 2010, Knodel et al. 2008]. Por exemplo, modelos de reflexão podem precisar de refinamentos sucessivos no modelo de componentes e linguagens de domínio específico podem requerer uma extensiva e detalhada especificação de restrições arquiteturais. Desta forma, em decorrência da complexidade em se especificar manualmente as restrições arquiteturais - o que invariavelmente tende a se revelar uma tarefa tediosa e sujeita a erros - técnicas tradicionais de conformidade arquitetural são algumas vezes rotuladas como inadequadas [Clements and Shaw 2009].

Para mitigar este problema, este artigo propõe ArchLint, uma ferramenta que combina técnicas de análise estática e histórica de código fonte para fornecer uma alternativa leve para verificação de conformidade arquitetural, a qual não requer refinamentos sucessivos e também não requer uma lista detalhada de restrições arquiteturais. Experimentos realizados com ArchLint mostram que é possível detectar ausências (falta de uma dependência) e divergências (dependências indesejadas) com precisão superior a 50%. Além disso, a abordagem de detecção é realizada estaticamente e de forma não-invasiva, não impactando atividades de desenvolvimento, manutenção e evolução do sistema.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta uma visão detalhada da abordagem proposta. A Seção 3 apresenta a arquitetura da ferramenta ArchLint. A Seção 4 reporta uma avaliação da aplicação de ArchLint em cinco sistemas reais. A Seção 5 discute trabalhos relacionados e a Seção 6 apresenta as conclusões.

2. ArchLint: Visão Geral

A Figura 1 ilustra a abordagem utilizada por ArchLint para detecção de violações arquiteturais. Basicamente, ela baseia-se em dois tipos de informações de entrada sobre o sistema alvo: (a) histórico de versões; (b) especificação dos componentes de alto nível. ArchLint considera que as classes do sistema estão estaticamente organizadas em módulos (ou pacotes, usando a terminologia Java) e que módulos estão logicamente agrupados em componentes. O modelo de componentes inclui informações para identificar os componentes, por meio de seus nomes, e um mapeamento dos módulos para os componentes definidos, usando expressões regulares. Dessa forma, ArchLint identifica dependências (ou a falta delas) suspeitas no código fonte baseando-se em hipóteses de frequência de uso e manutenções realizadas sobre essas dependências. ArchLint considera todas as dependências estáticas estabelecidas entre classes, incluindo aquelas relacionadas a chamadas de métodos, declarações de variáveis, herança, exceções, etc.

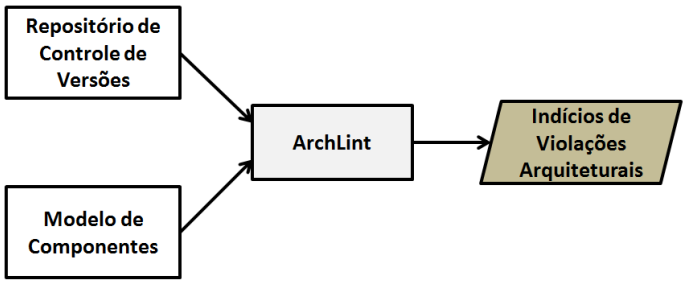

Figura 1. Abordagem proposta

ArchLint utiliza uma heurística para identificar ausências e três heurísticas para detectar divergências, descritas a seguir.

Heurística de Ausência: para detectar ausências, inicialmente procura-se por classes que denotam minorias no nível de componentes, em relação a uma dada dependência. Assumindo que as ausências são uma propriedade excepcional nas classes, minorias têm mais chances de representar violações arquiteturais. Além disso, o histórico de versões é usado para descobrir dependências dep introduzidas em classes originalmente criadas sem dep. O objetivo é reforçar as evidências (minorias) coletadas na etapa anterior, verificando se as classes originalmente criadas com a violação arquitetural em análise (por exemplo: ausência de dep) foram refatoradas mais tarde (por exemplo: introduziram dep). A suposição subjacente é que violações arquiteturais são geralmente detectadas e corrigidas.

Heurística de Divergência #1: essa heurística determina que a ocorrência de divergências deve ser restrita a dependências presentes em um pequeno número de classes de um determinado componente. Além disso, esta heurística inclui duas condições suplementares. Em primeiro lugar, estabelece-se que a dependência foi frequentemente removida do componente em análise - ou seja, a dependência foi caracterizada como violação e excluída do sistema. Em segundo lugar, a heurística também procura por outros componentes, onde a dependência é amplamente encontrada, por exemplo, componentes que se comportam como "*heavy-users*" da dependência sob análise. Supõe-se que é comum ter grupos de classes relacionadas e que, de acordo com a arquitetura planejada, só devem ser acessadas por classes localizadas em componentes bem delimitados.

Heurística de Divergência #2: tal como no caso anterior, esta heurística restringe a análise para dependências originadas a partir de um pequeno número de classes de um determinado componente e para dependências que foram removidas no passado. No entanto, ela apresenta duas diferenças importantes em relação à primeira heurística: (a) ela se baseia em dependências para uma classe alvo específica e não para módulos completos, (b) ela não requer a existência de um *heavy-user* para a dependência em análise.

Heurística de Divergência #3: essa heurística é baseada na suposição de que uma consequência comum das divergências é a criação de ciclos assimétricos entre componentes. A idéia é procurar pares de componentes cp_1 e cp_2 onde a maioria das referências é de cp_2 para cp_1. Assume-se que, na arquitetura original, esses componentes foram projetados para se comunicar de forma unidirecional e as dependências no "sentido errado" são de fato violações arquiteturais. Essa heurística é particularmente útil para detectar violações do tipo *back-call*, típicas em arquiteturas de camadas e que acontecem quando uma camada inferior utiliza de serviços implementados pelas camadas superiores.

3. Arquitetura Interna

Um protótipo da ferramenta ArchLint foi implementado para detectar violações arquiteturais. A Figura 2 apresenta a arquitetura interna da ferramenta. A implementação segue um padrão arquitetural em forma de *pipeline* com três componentes principais: *Extrator de Código*, *Extrator de Dependências* e *Detector de Violações Arquiteturais*.

O *Extrator de Código* é responsável pela extração (*checkout*) de todas as versões do sistema no repositório de controle de versões. Atualmente, nosso protótipo fornece suporte apenas a sistemas implementados em Java e acesso apenas a repositórios SVN.

O *Extrator de Dependências* é responsável pela criação de um modelo de dependências

que descreve relações estruturais existentes em cada versão do código fonte extraído. Para extrair as dependências do código-fonte, utiliza-se a ferramenta *VerveineJ*[1]. O modelo de dependências é então persistido em um banco de dados relacional.

O *Detector de Violações Arquiteturais* implementa as heurísticas descritas na Seção 2. Como as dependências são armazenadas em um banco de dados relacional, as heurísticas são implementadas por meio de consultas SQL.

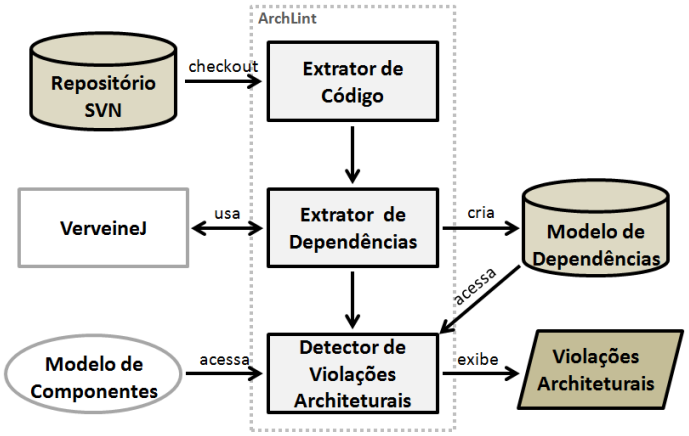

Figura 2. Arquitetura da ferramenta ArchLint

4. Aplicações

Em um trabalho anterior [Maffort et al. 2012], a ferramenta ArchLint foi avaliada em um sistema de informação utilizado por uma universidade brasileira. A Tabela 1 apresenta a precisão e *recall* alcançados neste primeiro estudo. Como pode ser observado, ArchLint alcançou precisão de 39,8% para ausências e 51,7% para divergências. Estes valores de precisão são compatíveis com os gerados, por exemplo, pelo FindBugs [Kim and Ernst 2007] e PMD, que são ferramentas conhecidas para detecção de *bugs* baseadas na análise estática.

Com o auxílio de informações e validações realizadas pelo arquiteto do sistema avaliado, para divergências, ArchLint alcançou um *recall* de 96,2%, não detectando apenas três divergências presentes no código fonte. Não foi possível medir o *recall* para ausências, pois seria necessário encontrar todo o conjunto de dependências que estão faltando, o que, na prática, requer uma inspeção detalhada e completa do código fonte.

Tabela 1. Primeiro estudo: precisão e recall

	Ausências	Divergências	Total
Indícios (I)	108	147	255
Verdadeiro Positivos (VP)	43	76	119
Falso Positivos (FP)	65	71	136
Falso Negativos (FN)	-	3	-
Precisão (VP/I)	39.8%	51.7%	46.7%
Recall (VP/(VP+FN))	-	96.2%	-

[1]https://gforge.inria.fr/projects/verveinej

Um segundo estudo foi realizado utilizando-se quatro sistemas *open-source*: Ant, ArgoUML, Lucene e SweetHome3D. A Tabela 2 exibe a precisão e o *recall* alcançados por ArchLint para divergências. Para os sistemas Ant, ArgoUML e Lucene a precisão foi superior a 57%. No caso do SweetHome3D, ArchLint não relatou um único verdadeiro positivo, mas o número de indícios também foi bastante pequeno. Em termos de *recall*, o maior resultado foi obtido para o sistema Ant (93%). No caso do SweetHome3D, não foi possível calcular o *recall* pois não foram detectadas divergências nesse sistema.

Tabela 2. Segundo estudo: precisão e recall

Sistema	Indícios	VP	FP	Precisão	FN	Recall
Ant	35	27	8	77.1%	2	93.1%
ArgoUML	42	24	18	57.1%	124	16.2%
Lucene	160	143	17	89.4%	169	45.8%
SweetHome3D	7	0	7	0.0%	0	-

5. Trabalhos Relacionados

Diversas ferramentas para extração de padrões de programação a partir de repositórios de software já foram propostas. DynaMine é uma ferramenta que analisa o código fonte para descobrir padrões específicos de codificação, como chamadas de método correlacionadas [Livshits and Zimmermann 2005]. BugMem [Kim et al. 2006] e FixWizard [Nguyen et al. 2010] são ferramentas que buscam por repetidas alterações de correção de *bugs* no histórico de versões de um projeto, por exemplo, mudanças nas quais uma condição incorreta é substituída por uma correta. Lamarck é uma ferramenta que busca por padrões de evolução em repositórios de software [Mileva et al. 2011]. Em Lamark, para avaliar a eficácia da ferramenta na detecção de erros, a precisão foi definida como: (#*code smells* e defeitos) / (#avisos emitidos pela ferramenta). Usando essa definição, a taxa de sucesso de Lamarck variou entre 33% e 64%. Em geral, essas ferramentas adotam uma abordagem vertical para descobrir padrões específicos do projeto em repositórios de software (em contraste com as ferramentas de análise estática que assumem uma abordagem horizontal baseada em um conjunto pré-definido de padrões de erros). ArchLint também utiliza uma abordagem vertical, mas com foco em conformidade arquitetural, ou seja, o objetivo é procurar por dependências estruturais que denotam ausências e divergências em relação à arquitetura estática de um sistema.

Em um trabalho recente, foram utilizadas regras de associação para extrair padrões arquiteturais [Maffort et al. 2013]. Em primeiro lugar, o objetivo foi investigar a geração automática de restrições arquiteturais na linguagem DCL [Terra and Valente 2009]. Em segundo lugar, procurou-se propor uma teoria para explicar e apoiar as heurísticas utilizadas no presente trabalho. Particularmente, foi possível concluir que a heurística para ausências e as duas primeiras heurísticas para divergências podem ser modeladas como um problema de mineração de itens frequentes.

6. Conclusão

No melhor conhecimento dos autores deste artigo, ArchLint é a primeira ferramenta de conformidade arquitetural que se baseia em uma combinação de técnicas de análise estática e histórica de código fonte. Neste artigo, foram apresentados os resultados da

aplicação de ArchLint em cinco sistemas reais, com precisão global variando de 46,7% até 89,4% e *recall* entre 16,2% até 93,1%. Como trabalho futuro, planeja-se investigar a aplicação de outras técnicas para a detecção de padrões arquiteturais, como a análise formal de conceitos. Além disso, pretende-se aplicar ArchLint em outros sistemas e ampliar a abordagem com novas heurísticas.

A ferramenta ArchLint está publicamente disponível em `http://java.labsoft.dcc.ufmg.br/archlint/`.

Agradecimentos: Este trabalho foi apoiado pela CAPES, FAPEMIG e CNPq.

Referências

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

[Kim and Ernst 2007] Kim, S. and Ernst, M. D. (2007). Which warnings should I fix first? In *15th International Symposium on Foundations of Software Engineering (FSE)*, pages 45–54.

[Kim et al. 2006] Kim, S., Pan, K., and Whitehead, Jr., E. E. J. (2006). Memories of bug fixes. In *14th International Symposium on Foundations of Software Engineering (FSE)*, pages 35–45.

[Knodel et al. 2008] Knodel, J., Muthig, D., Haury, U., and Meier, G. (2008). Architecture compliance checking - experiences from successful technology transfer to industry. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 43–52.

[Knodel and Popescu 2007] Knodel, J. and Popescu, D. (2007). A comparison of static architecture compliance checking approaches. In *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, page 12.

[Livshits and Zimmermann 2005] Livshits, B. and Zimmermann, T. (2005). DynaMine: finding common error patterns by mining software revision histories. In *13th International Symposium on Foundations of Software Engineering (FSE)*, pages 296–305.

[Maffort et al. 2012] Maffort, C., Valente, M. T., and Bigonha, M. (2012). Detecção de violações arquiteturais usando histórico de versões. In *XI Simpósio Brasileiro de Qualidade de Software (SBQS)*, pages 1–15.

[Maffort et al. 2013] Maffort, C., Valente, M. T., Bigonha, M., Hora, A., and Anquetil, N. (2013). Mining architectural patterns using association rules. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 375–380.

[Mileva et al. 2011] Mileva, Y. M., Wasylkowski, A., and Zeller, A. (2011). Mining evolution of object usage. In *25th European conference on Object-oriented programming*, pages 105–129.

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

[Nguyen et al. 2010] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J., and Nguyen, T. N. (2010). Recurring bug fixes in object-oriented programs. In *32nd International Conference on Software Engineering (ICSE)*, pages 315–324.

[Passos et al. 2010] Passos, L., Terra, R., Diniz, R., Valente, M. T., and Mendonca., N. (2010). Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89.

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