EDUARDO FERNANDO DE LIMA

ARCHPYTHON: VERIFICANDO ARQUITETURAS DE SISTEMAS PYTHON

LAVRAS – MG

2020

EDUARDO FERNANDO DE LIMA

ARCHPYTHON: VERIFICANDO ARQUITETURAS DE SISTEMAS PYTHON

Trabalho apresentado à Universidade Federal de Lavras a fim de cumprir os requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Prof. Dr. Ricardo Terra Nunes Bueno Villela
Orientador

LAVRAS – MG
2020

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Lima, Eduardo Fernando de.

ArchPython: Verificando Arquiteturas de Sistemas Python / Eduardo Fernando de Lima. – Lavras : UFLA, 2020.

44 p. :

TCC (graduação) –Universidade Federal de Lavras, 2020.

Orientador: Prof. Dr. Ricardo Terra Nunes Bueno Villela.

Bibliografia.

1. Arquitetura de Software. 2. Conformidade Arquitetural. 3. Visualização Arquitetural. I. Villela, Ricardo Terra Nunes Bueno. II. Título.

EDUARDO FERNANDO DE LIMA

ARCHPYTHON: VERIFICANDO ARQUITETURAS DE SISTEMAS PYTHON

Trabalho apresentado à Universidade Federal de Lavras a fim de cumprir os requisitos para a obtenção do título de Bacharel em Ciência da Computação.

APROVADA em 03 de setembro de 2020.

Rafael Serapilha Durelli	UFLA
Sergio Henrique Miranda Júnior	UFMG

Prof. Dr. Ricardo Terra Nunes Bueno Villela
Orientador

LAVRAS – MG
2020

Dedico este trabalho à minha família.

AGRADECIMENTOS

Agradeço ao Professor Ricardo Terra pela sua dedicação como orientador deste trabalho.

Agradeço à Universidade Federal de Lavras por disponibilizar recursos para a realização do trabalho.

RESUMO

Linguagens dinamicamente tipadas proveem uma série de recursos para os desenvolvedores, como invocações e construções dinâmicas. No entanto, ao se combinar tais recursos com prazos curtos, conflitos nos requisitos ou dificuldades técnicas, pode ocorrer com que a arquitetura de um sistema de software se distancie de sua arquitetura planejada, levando ao fenômeno denominado *erosão arquitetural*. Embora Python seja atualmente a 3ª linguagem de programação mais utilizada, não há uma ferramenta que permita os desenvolvedores a monitorar a arquitetura de seus sistemas. Isso se justifica, possivelmente, pela complexidade na inferência de tipos uma vez que uma mesma variável pode assumir diferentes tipos em tempo de execução. Diante desse desafio, este artigo propõe ArchPython, a primeira ferramenta completa de conformidade e visualização arquitetural para sistemas Python. Em suma, desenvolvedores especificam a arquitetura de seus sistemas forma simples e natural em arquivos JSON e ArchPython se encarrega do resto. De forma totalmente automática, tipos são inferidos (Jedi + heurística de propagação) e violações arquiteturais são detectadas (divergências, ausências e até mesmo alertas). Além de um relatório textual em formato JSON, a ferramenta ainda provê duas formas de visualização de violações arquiteturais (grafo e DSM).

Palavras-chave: Arquitetura de Software. Inferência de Tipos. Conformidade Arquitetural. Visualização Arquitetural.

ABSTRACT

Dynamically typed languages provide several resources for developers, such as dynamic invocations and constructs. However, such resources combined with short deadlines, conflicts in requirements, or technical difficulties may increase the number of code decisions that go against the planned architecture, leading to the phenomenon known as *architectural erosion*. Even though Python is the 3rd most used programming language, there is no tool that allows developers to monitor the architecture of their systems. This is possibly justified by the complexity to infer types since the same variable can assume different types at run time. Facing such challenge, this article proposes ArchPython, the first complete architectural conformance and visualization tool for Python systems. In a nutshell, developers specify the architecture of their systems in a simple and natural way using JSON files and ArchPython takes care of the rest. Automatically, the tool infers types (Jedi + propagation heuristics) and detects architectural violations (divergences, absences, and even alerts). In addition to a JSON-based textual report, the tool also provides two ways of visualizing architectural violations (graph and DSM).

Keywords: Software Architeture. Type Inference. Architetural Conformance. Architetural Visualization.

LISTA DE ABREVIATURAS SIGLAS

DAO	Data Access Object
DCL	Dependency Constraint Language
DSM	Dependency Structure Matrix
JSON	JavaScript Object Notation
IDE	Integrated Development Environment
MVC	Model-View-Controller

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

No início do desenvolvimento de software, é usual definir um projeto arquitetural para que o sistema de software possua alguns benefícios como manutenibilidade, escalabilidade, portabilidade, etc. (KNODEL et al., 2006; KNODEL; POPESCU, 2007; MURPHY; NOTKIN; SULLIVAN, 1995; PASSOS et al., 2009). No entanto, durante sua implementação, podem ocorrer desvios da arquitetura planejada devido a prazos curtos, requisitos conflitantes ou dificuldades técnicas. O acúmulo desses desvios leva ao fenômeno conhecido como *erosão arquitetural* (PERRY; WOLF, 1992) que – se negligenciado – pode transformar a arquitetura do sistema em um bloco monolítico com baixa coesão e alto acoplamento.

Este artigo é centrado na conjectura de que o processo de erosão arquitetural é ainda mais severo em linguagens dinamicamente tipadas por pelo menos duas razões: (i) alguns recursos providos por tais linguagens (e.g., invocações e construções dinâmicas, eval, etc.) tornam os desenvolvedores mais propícios a quebrar a arquitetura planejada e (ii) os ecossistemas de tais linguagens raramente ou mesmo não possuem ferramentas voltadas a aspectos arquiteturais. Por exemplo, Python é atualmente a 3ª linguagem de programação mais utilizada[1] e, mesmo assim, não há uma ferramenta que permita aos desenvolvedores monitorar a arquitetura de seus sistemas. Isso se justifica, possivelmente, pela complexidade na inferência de tipos uma vez que uma mesma variável pode assumir diferentes tipos em tempo de execução.

Diante desse desafio, este artigo propõe `ArchPython`, a primeira ferramenta completa de conformidade e visualização arquitetural para sistemas Python. Em suma, desenvolvedores especificam a arquitetura de seus sistemas de forma simples e natural em arquivos JSON definindo as dependências que um conjunto de arquivos pode ou não estabelecer dependência (*allowed* ou *forbidden*) e que de-

[1] TIOBE Index, https://www.tiobe.com/tiobe-index, em julho de 2020.

vem estabelecer dependência (*required*) e `ArchPython` se encarrega do resto. De forma totalmente automática, tipos são inferidos (Jedi + heurística de propagação) e violações arquiteturais são detectadas (divergências, ausências e até mesmo alertas). Além de um relatório textual em formato JSON, a ferramenta ainda provê duas formas de visualização de violações arquiteturais (grafo e DSM).

Este trabalho foi escrito originalmente no formato de artigo para submissão no XXXIV Simpósio Brasileiro de Engenharia de Software. Sendo assim, este trabalho é organizado como a seguir. O Capítulo 2 apresenta a ferramenta, descrevendo como especificar a arquitetura, como os tipos são inferidos e como as violações são detectadas e visualizadas. O Capítulo 3 descreve o projeto e implementação da ferramenta – e de seus módulos internos – com um viés mais técnico. O Capítulo 4 aborda as ferramentas relacionadas e o Capítulo 5 conclui.

2 FERRAMENTA

O ArchPython é uma ferramenta escrita em Python que realiza análise de conformidade arquitetural a partir de código estático. O objetivo da ferramenta é prover a desenvolvedores e arquitetos um meio de monitorar o processo de erosão arquitetural reportando violações (conformidade) e as apresentando de forma gráfica (visualização) para facilitar a observação das violações identificadas.

Conforme ilustrado na Figura 2.1, a partir do arquivo que especifica as regras da arquitetura e o caminho do código fonte do sistema a ser avaliado, a ferramenta:

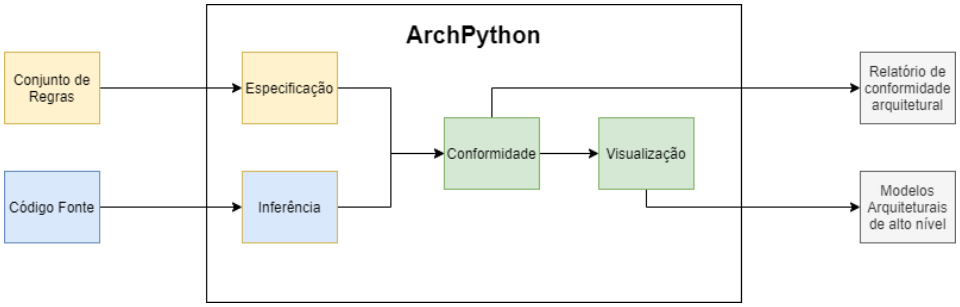
1. realiza o *parsing* do conjunto de regras (*Especificação*) que é detalhado na Subseção 2.2;

2. infere os tipos de todas as variáveis (*Inferência*) que é detalhado na Subseção 2.3;

3. verifica se os módulos do sistema estão de acordo com a arquitetura planejada (*Conformidade*) que é detalhado na Subseção 2.4; e

4. provê visualizações das violações arquiteturais (*Visualização*) que é detalhado na Subseção 2.5.

Figura 2.1 – Visão geral do ArchPython

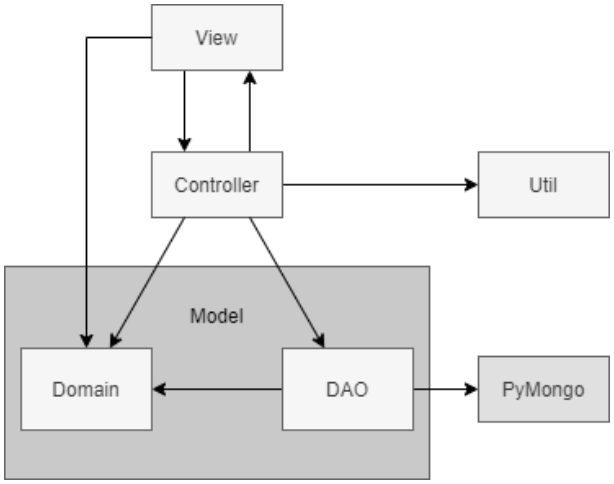


Como resultado, a solução oferece ao desenvolvedor um relatório textual com as violações e dois modelos arquiteturais de alto nível – um baseado em grafo e o outro em matriz – para uma melhor visualização dos problemas identificados.

2.1 *Running Example*

Para melhor conduzir esta seção, desenvolveu-se uma aplicação simples para gerenciamento de tarefas denominado **`PythonToDo`**. Conforme ilustrado na Figura 2.2, a aplicação segue o já conhecido padrão arquitetural *Model-View-Controller* (MVC) (FOWLER, 2002)[1] e possui como única dependência externa o pacote `PyMongo` para conexão com o banco de dados *MongoDB*.

Figura 2.2 – Arquitetura ideal da aplicação **`PythonToDo`**



Neste padrão, o componente `View` é composto por objetos que geralmente são associados a interface do usuário, como objetos que exibem textos, imagens, botões, janelas e campos de entrada. O `Model` é o componente do padrão utilizado

[1] O modelo MVC é comumente confundido com o modelo MVP (*Model-View-Presenter*). No modelo MVC, os objetos de domínio (dentro de *Model*) são passados para *View* através da *Controller*, logo, é permitido que a *View* utilize objetos de domínio. No modelo MVP, o *Presenter* transforma os objetos de domínio em objetos de apresentação de forma que a *View* nunca estabeleça contato direto com o *Model* (SYROMIATNIKOV; WEYNS, 2014).

para encapsular as informações da aplicação e, por fim, o componente `Controller` existe para mediar as interações entre o `View` e o `Model`.

No exemplo construído, o `Model` possui `Data Access Objects` (DAO) para encapsular o *framework* de persistência e `Domain Objects` que representam as entidades da aplicação, como *tasks*, *sub tasks* e *users*. Também existe um módulo `util` que possui os arquivos de utilidade.

2.2 Especificação

A especificação das regras da arquitetura são declaradas no formato JSON. O motivo dessa escolha deve-se a facilidade da declaração e leitura das regras por qualquer desenvolvedor; e devido a sua popularidade no ecossistema Python por conta de possuir um módulo JSON em sua biblioteca padrão (FOUNDATION, 2020b). O arquivo de entrada possui um arranjo de elementos que segue a estrutura descrita na Listagem 2.1. Na especificação, as linhas que possuem `optional` no final, são linhas que tem sua declaração opcional e expressões sucedidas pelo símbolo + representam que podem ser declaradas uma ou mais vezes.

```
1 "ModuleId":  {
2   "files or packages": [ Pattern+ ]
3   "allowed or forbidden": [ ModuleId+ ] #optional
4   "required": [ ModuleId+ ] #optional
5 }
```

Listagem 2.1 – Estrutura da declaração de um módulo

O **`ModuleId`** representa o nome do módulo declarado. Um módulo é obrigatoriamente composto por arquivos (**`files`**) ou por pacotes (**`packages`**) sucedidos de um arranjo que indica todos os caminhos de arquivos ou nomes dos pacotes (representados por `Pattern` na especificação) que pertencem ao módulo, sendo impossível combinar pacotes e arquivos na mesma definição.

Em seguida, podem ser declarados os módulos que são permitidos (**`allowed`**) ou proibidos (**`forbidden`**) de serem acessados. Também não é pos-

sível combinar as definições de permitidos e proibidos, o `ArchPython` considera que os pacotes não declarados em um item pertencerão ao espectro oposto (i.e., todos os módulos que não foram declarados como permitidos são considerados proibidos). Por fim, de modo opcional, são declarados os módulos que devem ser acessados, i.e., a dependência é obrigatória (**`required`**).

Para ilustrar uma especificação JSON, a Listagem 2.2 apresenta as regras que definem a arquitetura do `PythonToDo`. Por exemplo, entre as linhas 2 e 6, é indicado que o módulo `DAO` permite o uso de arquivos do módulo `Domain` e é obrigado a utilizar os pacotes definidos pelo `PyMongo`. De forma similar, são proibidos aos arquivos que pertencem ao módulo `Controller` de utilizarem o `PyMongo` (linha 10), visto que, na arquitetura MVC, um `Controller` não deve possuir e nem acessar diretamente uma instância do banco de dados. O `ArchPython` é capaz de reconhecer expressões regulares na declaração dos caminhos dos arquivos, permitindo que seja possível, por exemplo, declarar que os arquivos que compõem o módulo `Controller` (linha 8) são aqueles que terminam com ".py" e que estão dentro da pasta `controllers`.

```
1 {
2   "DAO": {
3     "files": ["./db/*.py", "./models/dao/*.py"],
4     "allowed": ["Domain"],
5     "required": ["PyMongo"]
6   },
7   "Controller": {
8     "files": ["./controllers/*.py"],
9     "required" : ["View"],
10    "forbidden": ["PyMongo"]
11  },
12  "View": {
13    "files": ["./views/*.py"],
14    "allowed": ["Domain"]
15  },
16  "Domain": {
17    "files": ["./models/domain_objects/*.py"]
18  },
19  "Util" : {
20    "files": ["./util/*.py"]
```

```
21   },
22   "PyMongo": {
23     "packages": ["pymongo", "bson"]
24   }
25 }
```

Listagem 2.2 – Especificação do *Running Example*

As demais especificações possuem as mesmas construções salvo o módulo `PyMongo` que é composto pelos pacotes `pymongo` e `bson` (linha 23), diferentemente da maioria que utiliza "`files`".

Para evitar possíveis erros de digitação, o ArchPython alertará ao usuário definições declaradas diferentes de **`files`**, **`packages`**, **`allowed`**, **`forbidden`** e **`required`**. Isso evita que o usuário incorretamente escreva "forbiden"e acredite que o módulo não tem nenhuma violação, uma vez que nem estaria verificando.

2.3 Inferência de Tipos

A inferência de tipos foi construída com base no trabalho de Miranda et al. (MIRANDA et al., 2016), porém com aprimoramentos possíveis por meio de bibliotecas populares em Python (HALTER, 2012).

Em poucas palavras, a heurística utilizada constrói em um primeiro momento um conjunto `TIPOS` no qual cada elemento é um tripla [`metodo`,`nome_var`,`tipo`], sendo que `tipo` indica o possível tipo da variável identificada por `nome_var` que foi instanciada dentro da função denominada `metodo`.

Dessa forma, esse conjunto é definido recursivamente como a seguir:

- **Passo Base:** Para cada instanciação direta do tipo `T` atribuída a uma variável `x` dentro de método `f`, [`f`,`x`,`T`] $\in$ `TIPOS`[2]. Além disso, para cada invoca-

[2] Por definição, no caso em que uma classe X acessa um método foo de uma classe B, porém que foi herdado de uma classe A, é considerado que X estabelece uma dependência com B e não com A.

ção de uma função g() cujo retorno é atribuído a uma variável x dentro do método f, $\forall T \in retorno(g)$, [f,x,T] $\in$ TIPOS.

- **Passo Recursivo:** Se [f,x,T] $\in$ TIPOS e existe uma chamada g(x) dentro de f, então [g,y,T] $\in$ TIPOS, onde y é o nome do parâmetro formal em g. Esse passo é aplicado até nenhuma nova tripla seja adicionada ao conjunto TIPOS.

O código na listagem 2.3 será utilizado para exemplificar uma execução da heurística. Ao final do passo base, tem-se o conjunto TIPOS com os elementos [A::f,n,int], [A::f,a,Foo], [A::f,b,B], [A::f,r,Bar], [A::f,r,Qux], [B::g,c,C] e [C::h,d,D]. Basicamente, são todas as variáveis que (i) foram inicializadas diretamente ou (ii) que foram inicializadas a partir do retorno de uma função.

Em seguida, é iniciada a primeira execução do passo recursivo, as tuplas [B::g,x,Foo], [B::g,y,B], [B::g,z,int] são incluídas em TIPOS. Na segunda execução são adicionadas as tuplas [C::h,w,Foo] e [C::h,w,B]. Na terceira execução os tipos [D::m,k,Foo] e [D::m,k,B] são adicionados. Na quarta execução nenhuma nova tripla é adicionada a TIPOS, logo, o critério de parada é atendido e a execução da heurística termina.

```
1 class A:                          class B:
2   def f(self):                      def g(self, x, y, z):
3     n  = randint(0, 101)              c = C()
4     a = Foo()                         c.h(x)
5     b = B()                           c.h(y)
6     r = b.g(a, b, n)                  if (z % 2 == 0):
7                                         return Bar()
8                                       else:
9                                         return Qux()
10
11 class C:                         class D:
12   def h(self, w):                   def m(self, k):
13     d = D()                           print(k)
14     d.m(w)
```

Listagem 2.3 – Código exemplo para inferência de tipos

A partir do exemplo, pode-se ressaltar que o mecanismo de inferência (i) é capaz de identificar retorno de funções, (ii) considera os possíveis tipos que um parâmetro formal pode assumir através da propagação de tipos e, por consequência, (iii) melhora a eficácia da conformidade arquitetural por considerar todos os possíveis tipos durante a detecção de violações arquiteturais.

A Tabela 2.1 indica a quantidade de inferências realizadas sobre os parâmetros formais e as variáveis da aplicação `PythonToDo`. A ferramenta proposta pôde inferir pelo menos 70% das variáveis. Como limitação, não foi possível inferir os tipos retornados diretamente do banco de dados. Acredita-se que, por se tratar de tipos primitivos, isso não impactaria o processo de conformidade arquitetural. O processo de cálculo das variáveis inferidas foi simples. Com o uso da biblioteca Jedi (HALTER, 2012), obteve-se todas as variáveis declaradas no sistema alvo. Em seguida, realizou-se a comparação de todas as variáveis inferidas com todas as variáveis do sistema.

Table 2.1 – Inferências realizadas pelo `ArchPython`.

Objeto	Total	Inferido	Taxa de Inferência
Variáveis	70	50	71%
Parâmetros	20	15	75%

Logo após a inferência de tipos, a arquitetura atual do sistema alvo é extraída e a ferramenta provê um grafo com a arquitetura atual do sistema.

2.4 Conformidade Arquitetural

Esse processo possui como entrada as regras arquiteturais processadas e todos os tipos que cada módulo utiliza detectados por meio da aplicação da heurística. O primeiro passo é calcular quantas chamadas existem de um módulo A para um módulo B. Em seguida, são verificados separadamente os seguintes casos:

- **Convergência**: Existe um módulo A que possui arquivos que utilizam o módulo B e foi (i) declarado explicitamente que A poderia utilizar B ou (ii) não foi declarado explicitamente que A é proibido de utilizar B.

- **Divergência**: Existe um módulo A que possui arquivos que utilizam o módulo B, entretanto, foi (i) declarado explicitamente que A não é permitido utilizar o módulo B ou (ii) não foi explicitamente declarado que o módulo A poderia utilizar o módulo B

- **Ausência**: Existe um módulo A que possui arquivos que não utilizam o módulo B, sendo que foi declarado explicitamente que o Módulo A deve utilizar o Módulo B.

- **Alerta**: Existe um módulo A que possui arquivos que não utilizam o módulo B, entretanto, foi declarado que o módulo A poderia utilizar o módulo B. É importante destacar que alerta não é de fato uma violação, porém acredita-se que é importante destacar esse comportamento para que o desenvolvedor possa revisar a especificação arquitetural de seus sistemas.

Para exemplificar, foram inseridas no `PythonToDo` as seguintes duas violações que poderiam acontecer em um caso real e que não alteraria o funcionamento da aplicação:

- **Ausência**: Foi declarado que o módulo `DAO` requer a utilização do pacote `PyMongo`. Entretanto, foi criado um arquivo dentro do módulo `DAO` que não utiliza o `PyMongo`.

- **Divergência**: Foi informado que o módulo `Controller` é proibido de utilizar diretamente o pacote `PyMongo`. Todavia, a `controller` de Tarefas (`TaskController`) possui uma operação de listagem que é realizada dentro da `controller`.

Ao final da conformidade arquitetural, conforme reportada na Listagem 2.4, a ferramenta provê um arquivo no formato JSON que, por padrão, reporta cada ocorrência que é considerada uma violação, ou seja, todas as ocorrências que são divergências ou ausências. É possível que os alertas sejam incluídos no arquivo JSON. Para isso, basta o usuário informar o argumento "`--report-conformity-warnings`" ao executar a ferramenta.

```
{
 "DIVERGENCE": [
   {
    "origin_module": "Controller",
    "origin_line": 32,
    "file_path": "./controller/TaskController.py",
    "target_module": "pymongo",
    "constraint": "Controller cannot depend on pymongo"
   }
  ],
 "ABSENCE": [
   {
    "origin_module": "DAO",
    "origin_line": "",
    "file_path": "./model/dao/SubTaskDAO.py",
    "target_module": "pymongo",
    "constraint": "DAO does not depend on pymongo"
   }
  ]
}
```

Listagem 2.4 – Relatório JSON das violações arquiteturais

2.5 Visualização

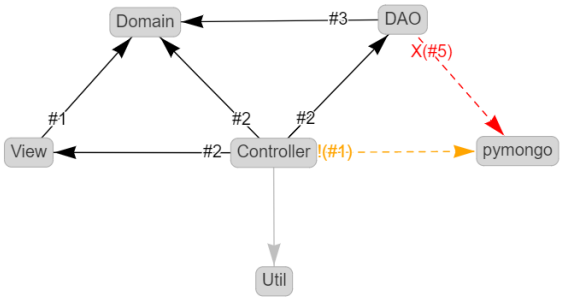
Após o processo de conformidade arquitetural, a ferramenta `ArchPython` apresenta para o usuário duas formas de visualizar as violações arquiteturais. Foram utilizadas cores para representar a situação do relacionamento entre dois módulos. A cor *preta* representa convergência, a cor *laranja* junto com o símbolo "!"

simboliza divergência, a cor *vermelha* junto com o símbolo "X" significa ausência e, por último, a cor *cinza* representa um alerta.

O primeiro modelo é um grafo de dependências orientado onde cada nó simboliza um módulo do sistema e cada aresta possui um número que indica a quantidade de chamadas entre os módulos e uma cor que varia de acordo com a violação.

A Figura 2.3 ilustra o grafo de dependências gerado pelo `ArchPython` utilizando como entrada o sistema motivador.

Figura 2.3 – Grafo de reflexão gerado pelo ArchPython



O grafo construído foi baseado no modelo de reflexão proposto por Murphy et al (MURPHY; NOTKIN; SULLIVAN, 1995). O modelo de Murphy combina um modelo de alto nível definido pelo arquiteto com um modelo de baixo nível, que representa o estado atual da arquitetura. Essa combinação resulta em um grafo que destaca as divergências e as ausências da arquitetura.

Entretanto, grafo, por natureza, não é uma forma de visualização escalonável (MIRANDA et al., 2016) tornando-se bastante confuso caso um sistema de grande porte seja avaliado. Para suprir isso, o `ArchPython` também fornece a visualização por meio de uma Matriz de Dependência Estrutural (DSM).

A Figura 2.4 ilustra a DSM gerada pelo `ArchPython` utilizando como entrada o sistema motivador.

Figura 2.4 – DSM gerada pelo `ArchPython`

Modules		1	2	3	4	5	6
Controller	**1**						
DAO	**2**	2					
Domain	**3**	2	3			1	
Util	**4**	?					
View	**5**	2					
pymongo	**6**	1	5				

DSM foi inicialmente proposta para demonstrações na indústria de hardware (BALDWIN et al., 2000), entretanto, Sullivan et al. (SULLIVAN et al., 2001) observaram que o conceito também poderia ser utilizado na indústria de software. A DSM é uma matriz quadrada na qual cada célula indica a quantidade de vezes que o módulo indicado pela coluna realizou uma chamada no módulo indicado pela linha. As células que possuem violações arquiteturais foram destacadas com as cores descritas anteriormente.

De forma sumarizada, é possível observar as duas violações inseridas propositalmente. A ausência é representada pela aresta vermelha "X(#1)" (grafo) e pela célula vermelha com conteúdo "1" (DSM), indicando que uma classe `DAO` não utiliza `PyMongo` mesmo sendo obrigatório o estabelecimento de tal dependência. Enquanto que a divergência é representada pela aresta laranja "!(#1)" (grafo) e pela célula laranja com conteúdo "1" (DSM) indicando que a classe `Controller` acessa uma vez `PyMongo` mesmo não sendo permitido.

3 PROJETO E IMPLEMENTAÇÃO

Esta seção apresenta o projeto e a implementação dos quatro módulos internos da ferramenta `ArchPython`: especificação, inferência de tipos, conformidade e visualização.

3.1 Módulo de Especificação

O módulo de especificação é o primeiro módulo a ser executado pela ferramenta e possui como entrada um arquivo no formato JSON declarando as restrições da arquitetura conforme Seção 2.2. Durante a leitura do arquivo de entrada, é verificado se um módulo não é constituído tanto por pacotes externos e arquivos e se não possui restrições de proibição e permissão ao mesmo tempo.

Este módulo utiliza algumas bibliotecas padrão da linguagem Python, como *json* (FOUNDATION, 2020b) para realizar a decodificação do arquivo de entrada e a biblioteca *glob* (FOUNDATION, 2020a) para reconhecer expressões regulares na declaração dos caminhos dos arquivos.

Desta forma, após realizar a decodificação do arquivo de entrada, este módulo provê uma *lista com os módulos e suas restrições*.

3.2 Módulo de Inferência de Tipos

Este módulo recebe como entrada o caminho do projeto a ser verificado e, em seguida, constrói uma lista com todos os arquivos Python dentro do diretório. Com a lista de arquivos, Jedi (HALTER, 2012) – uma popular biblioteca de análise estática de código Python amplamente utilizada neste módulo – realiza a análise de cada arquivo individualmente.

Dessa forma, ao aplicar a biblioteca sobre um arquivo Python, Jedi retorna uma lista de objetos *Name* que podem representar declarações, importações e estruturas da linguagem Python. Jedi também possui a funcionalidade de realizar

uma inferência em um objeto *Name* e retornar todos os possíveis tipos do objeto. Sendo assim, ao analisar um arquivo, identifica-se todos os objetos *Name* que são declarações de variáveis e realiza-se a inferência sobre tal objeto. Em paralelo, são registradas todas as chamadas de funções para a execução da heurística.

A biblioteca Jedi foi capaz de realizar inferências complexas, como inferir o retorno de funções de bibliotecas externas e funções embutidas da linguagem. Entretanto, a Jedi falhou em inferir o tipo das variáveis que são parâmetros formais. Portanto, para preencher essa lacuna, itera-se sobre a lista de chamadas de função para realizar a identificação dos parâmetros formais, conforme previamente descrito na Seção 2.3. Dessa forma, este módulo provê *uma lista com todas as variáveis e os tipos que podem assumir*.

3.3 Módulo de Conformidade Arquitetural

Este módulo recebe como entrada a "lista com os módulos e suas restrições" e "uma lista com todas as variáveis e os tipos que podem assumir".

Inicialmente, cria-se uma estrutura que armazena informações sobre a relação entre dois módulos, tal como quantidade de chamadas entre os módulos (sendo origem ou alvo da dependência) e um *status* que pode receber *ALLOWED*, *DIVERGENCE*, *ABSENCE* ou *WARNING*.

Assim, é averiguada a quantidade de interações entre dois módulos através da lista de inferências. Em seguida, a definição da arquitetura é utilizada em conjunto com a quantidade de iterações para definir o *status* da relação entre dois módulos. Como exemplo, o *status* de *DIVERGENCE* é atribuído a uma relação entre dois módulos quando a arquitetura diz que a relação entre os dois é proibida e existe um número de interações $n > 1$.

Após realizar as verificações, este módulo provê *uma lista de objetos com informações sobre as relações entre os módulos*. As relações que possuírem o *status DIVERGENCE* e *ABSENCE* são reportadas com detalhes em um arquivo JSON.

3.4 Módulo de Visualização

Este módulo provê duas formas de visualização das violações arquiteturais detectadas: um grafo de dependências e uma matriz de dependência estrutural. Ambas as formas de visualização foram construídas como páginas HTML.

As duas visualizações foram criadas em cima de *templates* pré-definidos pelos autores. De forma geral, o ArchPython processa os dados vindos do módulo de conformidade para cada tipo de visualização, lê o arquivo *template*, insere os dados e salva em um novo arquivo.

O módulo *network* da biblioteca vis.js é utilizado para construir o grafo de dependências e recebe como entrada um objeto JSON que define os nós, arestas, rótulos e cores. A matriz de dependência foi criada a partir de tabelas HTML, JavaScript e CSS, sem a utilização de dependências externas.

Limitações da Ferramenta

A maioria das limitações estão relacionadas às limitações do Jedi[1]. Entretanto, é importante clarificar que inferência de tipos é algo intrinsecamente complexo e *não* é o único propósito do Jedi. Mesmo assim, ArchPython, por conta de sua solidez e maturidade, utiliza uma ampla gama de funcionalidades do Jedi que podem ser indiretamente utilizadas para o propósito de inferência de tipos.

Por exemplo, o Jedi é capaz de realizar inferências não triviais, como inferir atributos de classes, retorno de funções estáticas e funções que não possuíam nenhum retorno, porém ainda não é capaz de inferir tipos de variáveis globais. Algumas outras limitações são tratadas através do passo recursivo da inferência de tipos, como objetos passados para funções que apenas repassam o objeto sem a realização de quaisquer acessos.

[1] O Jedi (HALTER, 2012) é uma ferramenta de análise estática muito empregada no projeto e implementação de diversos *plug-ins* de editores de texto e IDEs, sendo inclusive utilizada no *plug-in* oficial de Python (mantido pela Microsoft) do editor de texto Visual Studio Code, que atualmente conta com 23 milhões de downloads.

4 FERRAMENTAS RELACIONADAS

Atualmente, não é de conhecimento dos autores a existência de ferramentas para verificação de conformidade arquitetural de sistemas escritos em Python.

A solução proposta neste artigo teve inspiração no ArchRuby (MIRANDA et al., 2016) que, por sua vez, teve inspiração na ferramenta DCL Suite (TERRA; VALENTE, 2009), que é uma ferramenta de conformidade arquitetural integrada ao IDE Eclipse e que provê inclusive recomendações de reparo de violações (TERRA et al., 2015). A escolha do ArchRuby como inspiração ocorreu pelo fato de a ferramenta analisar sistemas escritos em Ruby que, assim como o Python, são linguagens de propósito geral, interpretadas e dinamicamente tipadas. A precisão da inferência de tipos realizada pelo ArchPython tende a ser melhor que a precisão da inferência realizada pelo ArchRuby. Isso acontece por conta do passo base do ArchPython realizar um número maior de inferências devido à utilização do Jedi.

A biblioteca Jedi (HALTER, 2012) é conhecida por possuir múltiplas funcionalidades de análise estática de código, sendo utilizada para construção de diversos *plug-ins* para editores de textos famosos como *Vim*, *Emacs* e *Visual Studio Code*. Conforme mencionado na Seção 2.3, Jedi é largamente utilizado na ferramenta proposta neste artigo, principalmente para a inferência de tipos. De forma complementar, durante o desenvolvimento deste trabalho, foi descoberto um *bug* na biblioteca na qual não era possível inferir todos os possíveis retornos de uma função. O *bug* foi reportado em 29/03/2020[1], o qual foi corrigido pela equipe de desenvolvimento do Jedi em menos de dois dias.

Existem poucas soluções com o objetivo de visualizar arquitetura de uma aplicação escrita em Python. O IDE *PyCharm* (JETBRAINS, 2000) disponibiliza uma *feature* que possibilita verificar a hierarquia do código fonte (e.g., quais os métodos que existem dentro de uma determinada classe ou quais classes são

[1] <https://github.com/davidhalter/jedi/issues/1530>

herdadas por determinada classe). No entanto, *PyCharm* é uma solução proprietária com código fechado. A ferramenta *SourceTrail* (COATISOFTWARE, 2015) também possibilita a visualização em alto nível do código fonte, sendo uma ferramenta de código aberto e gratuita. Entretanto, nos dias atuais, o suporte para Python permanece em fase beta e carece de testes.

Na literatura, Maia et al. propõem uma ferramenta para análise estática de códigos escritos em um subconjunto de Python denominado RPython (MAIA; MOREIRA; REIS, 2012). A partir de um conjunto de regras predefinidas, a ferramenta garante que nenhuma variável receberá valores de tipos diferentes durante a execução do programa e que todos os tipos complexos são homogêneos (i.e., todos os elementos de uma lista possuem o mesmo tipo; todas as chaves de um dicionários são do mesmo tipo). Vitousek argumenta que a tipagem gradual – que permite partes do programa serem analisadas de forma estática e outras de modo dinâmica – pode ter um desempenho ruim por conta de algumas características do Python (VITOUSEK, 2019). Com objetivo de melhorar o desempenho, o autor utiliza tipagem estática para otimizar o código e melhorar o desempenho da tipagem gradual. Monat et al. propõem uma ferramenta que realiza análise estática de código em busca de possíveis exceções que podem ser lançadas durante a execução do programa (MONAT; OUADJAOUT; MINÉ, 2020). Entretanto, o trabalho teve problemas ao detectar as possíveis exceções que poderiam ser lançadas pela biblioteca padrão do Python (por conta de algumas partes serem escritas na linguagem C).

5 CONSIDERAÇÕES FINAIS

Erosão arquitetural é um problema recorrente que afeta de forma negativa a manutenibilidade e a escalabilidade de um projeto. Esse problema é ainda mais grave em uma linguagem dinamicamente tipada como o Python pois (i) alguns de seus recursos tornam os desenvolvedores mais propícios a quebrar a arquitetura planejada, por exemplo, invocações e construções dinâmicas e (ii) o ecossistema da linguagem não possui até então ferramentas voltadas ao monitoramento do projeto arquitetural.

Diante desse cenário e pelo fato de Python ser uma das linguagens mais utilizadas no mercado e de possuir um extenso repositório de pacotes, este artigo propõe – e evidencia a necessidade de – uma ferramenta que realize a verificação da conformidade arquitetural de uma aplicação.

De forma sucinta, `ArchPython` contribui diretamente para a comunidade Python ao prover:

1. uma forma natural e simples de se especificar a arquitetura de um sistema (Seção 2.2);

2. uma heurística eficaz (Jedi + propagação) de inferência de tipos (Seção 2.3);

3. um mecanismo para detecção de violações e alertas arquiteturais (Seção 2.4); e

4. duas visualizações (grafo e DSM) de violações e alerts arquiteturais (Seção 2.5).

Mais importante, foi desenvolvida na Seção 2.1 uma aplicação de pequeno porte – `PythonToDo` – que ilustra cada uma das contribuições supracitadas no decorrer do texto.

O código do ArchPython – e também do `PythonToDo` – estão publicamente disponíveis sob a licença MIT em:

https://github.com/PqES/ArchPython/

Trabalhos futuros incluem:

1. Incluir um catálogo de padrões com notações visuais para gerar a especificação;
2. Incorporar o ArchPython a um editor de texto (e.g., Visual Studio Code);
3. Criar uma solução com sugestões de reparo para as violações encontradas;
4. Aprimorar a visualização do grafo incluindo uma funcionalidade que amplia e realça as relações de um módulo selecionado pelo usuário; e
5. Executar a ferramenta em um sistema de grande porte.

REFERÊNCIAS

BALDWIN, C. Y. et al. **Design rules: The power of modularity**. [S.l.]: MIT press, 2000. v. 1.

COATISOFTWARE. **SourceTrail**. 2015. Disponível em: <https://www.sourcetrail.com/>.

FOUNDATION, P. S. **glob - Unix style pathname pattern expansion**. 2020. Disponível em: <https://docs.python.org/3/library/glob.html>.

FOUNDATION, P. S. **json - JSON encoder and decoder**. 2020. Disponível em: <https://docs.python.org/3/library/json.html>.

FOWLER, M. **Patterns of enterprise application architecture**. 1. ed. [S.l.]: Addison-Wesley Longman Publishing., 2002.

HALTER, D. **Jedi**. 2012. Disponível em: <https://github.com/davidhalter/jedi>.

JETBRAINS. **PyCharm**. 2000. Disponível em: <https://www.jetbrains.com/pycharm/>.

KNODEL, J. et al. Static evaluation of software architectures. In: **10th European Conference on Software Maintenance and Reengineering (CSMR)**. [S.l.: s.n.], 2006. p. 279–294.

KNODEL, J.; POPESCU, D. A comparison of static architecture compliance checking approaches. In: **6th Working IEEE/IFIP Conference on Software Architecture (WICSA)**. [S.l.: s.n.], 2007. p. 12.

MAIA, E.; MOREIRA, N.; REIS, R. A static type inference for python. **Proc. of DYLA**, v. 5, n. 1, p. 1, 2012.

MIRANDA, S. et al. Architecture conformance checking in dynamically typed languages. **Journal of Object Technology**, v. 15, n. 3, p. 1–1, 2016.

MONAT, R.; OUADJAOUT, A.; MINÉ, A. Static type analysis by abstract interpretation of python programs. **ECOOP (LIPIcs). To appear**, 2020.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. Software reflexion models: Bridging the gap between source and high-level models. In: **Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering**. [S.l.: s.n.], 1995. p. 18–28.

PASSOS, L. et al. Static architecture-conformance checking: An illustrative overview. **IEEE software**, IEEE, v. 27, n. 5, p. 82–89, 2009.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **Software Engineering Notes**, v. 17, n. 4, p. 40–52, 1992.

SULLIVAN, K. J. et al. The structure and value of modularity in software design. **ACM SIGSOFT Software Engineering Notes**, ACM New York, NY, USA, v. 26, n. 5, p. 99–108, 2001.

SYROMIATNIKOV, A.; WEYNS, D. A journey through the land of model-view-design patterns. In: IEEE. **2014 IEEE/IFIP Conference on Software Architecture**. [S.l.], 2014. p. 21–30.

TERRA, R.; VALENTE, M. T. A dependency constraint language to manage object-oriented software architectures. **Software: Practice and Experience**, v. 39, n. 12, p. 1073–1094, 2009.

TERRA, R. et al. A recommendation system for repairing violations detected by static architecture conformance checking. **Software: Practice and Experience**, Wiley Online Library, v. 45, n. 3, p. 315–342, 2015.

VITOUSEK, M. M. Gradual typing for python, unguarded. [Bloomington, Ind.]: Indiana University, 2019.