Programa de Pós-Graduação em

Computação Aplicada

Mestrado Acadêmico

Lucas Silveira Kupssinskü

MITRAS: Modelo Inteligente para Transformação de Aplicações de Software

São Leopoldo, 2019

Lucas Silveira Kupssinskü

MITRAS:
Modelo Inteligente para Transformação de Aplicações de Software

Dissertação apresentada como requisito parcial para a obtenção do título de Mestre pelo Programa de Pós-Graduação em Computação Aplicada da Universidade do Vale do Rio dos Sinos — UNISINOS

Orientador:
Prof. Dr. João Carlos Gluz

São Leopoldo
2019

K96m Kupssinskü, Lucas Silveira.
MITRAS : modelo inteligente para transformação de aplicações de software / Lucas Silveira Kupssinskü. – 2019.
94 f. : il. ; 30 cm.

Dissertação (mestre) – Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, 2019.
"Orientador: Prof. Dr. João Carlos Gluz."

1. Transformação de grafo. 2. Software - Manutenção. 3. Processamento de linguagem natural. I. Título.
CDU 004

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecária: Amanda Schuster – CRB 10/2517)

Lucas Silveira Kupssinskü

MITRAS
Modelo Inteligente para Transformação de Aplicações de Software

Dissertação apresentada à Universidade do Vale do Rio dos Sinos – Unisinos, como requisito parcial para obtenção do título de Mestre em Computação Aplicada.

Aprovado em 07/03/2019

BANCA EXAMINADORA

Prof. Dr. João Carlos Gluz – Unisinos

Prof. Dr. Álvaro Freitas Moreira – UFRGS

Prof. Dr. Kleinner Silva Farias de Oliveira – Unisinos

Prof. Dr.João Carlos Gluz (Orientador)

São Leopoldo,

Prof. Dr. Rodrigo da Rosa Righi
Coordenador PPG em Computação Aplicada

RESUMO

Manutenção de sistemas é uma atividade custosa tanto para desenvolvedores quanto para usuários finais. Dessa forma, novas ideias e tecnologias se fazem necessárias para diminuir o esforço empregado nesse tipo de atividade. Essa dissertação realiza a apresentação do sistema MITRAS, que usa um modelo inteligente para transformação de aplicações de software visando fornecer a usuários sem experiência em desenvolvimento a capacidade de realizar modificações em sistemas a partir de uma explanação em linguagem natural.

Isto é possível combinando de uma nova forma ferramentas encontradas no estado da arte para processamento de linguagem natural e transformação de grafos aplicadas a modelos de software.

O MITRAS trabalha com manutenções perfectivas e adaptativas (conforme classificação adotada pela IEEE). Esses tipos de manutenção não ocorrem devido a erros, mas em melhorias identificadas por usuários em sistemas que já estão em funcionamento.

As solicitações do usuário e as respectivas modificações foram mapeadas em um modelo de grafos que permite a aplicação de conceitos de transformação de grafos para modelar e operacionalizar modificações de sistema. O sincronismo entre modelo de grafos e código fonte é abordado através de um processo de extração automatizada do modelo e de um processo de injeção diferencial de código fonte.

Esse trabalho apresenta tanto a base teórica formal do modelo quanto aspectos práticos de um protótipo desenvolvido em um repositório *open source* real. Os resultados do modelo são avaliados de duas formas: em laboratório para aferir tempo de execução e complexidade do sistema; e via experimentos com usuários, de modo a comparar as modificações feitas com auxílio do MITRAS com as modificações realizadas por um desenvolvedor. Os experimentos realizados indicaram que desenvolvedores e usuários conseguiram realizar as alterações propostas em 92% dos casos, porém usuários com acesso ao MITRAS levaram em média 8 minutos enquanto desenvolvedores levar em média 26 minutos.

Palavras-chave: Transformações de Grafo. Manutenção de Software. Processamento de Linguagem Natural.

ABSTRACT

Software maintenance is costly for developers and customers as well. In this regard, new ideas and technologies are necessary to minimize the effort in this type of activity. This dissertation presents MITRAS project, an intelligent model for software transformation that enables users without programming knowledge to do modifications in systems via interaction in natural language.

This is made possible combining state of the art tools and techniques in natural language processing and graph transformations applied to software models.

MITRAS is designed to work with adaptive and perfective type of maintenances (as classified by IEEE), this maintenance do not imply that the system has an error, but that there is need for improvements identified by the user in already deployed systems

User requests and the respective modifications are mapped in graphs that can be used in a graph transformation process that models system modifications. Synchronism between graph model and source code is maintained through automatic model extraction and differential code injection.

This work presents both formal theoretical foundations and practical aspects of MITRAS computational model and a prototype developed and tested against a real open source Project. The model results are assessed using two methods, the first one aims to check execution time and complexity of the system, the second one compares the results of a user of MITRAS and software developers. The experiments are presenting good results so far, indicating that users can do simple maintenance tasks with MITRAS in less time then developers but with similar correctness.

Keywords: Graph Transformation. Software Maintenance. Natural Language Processing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

SPO	Single Pushout
IA	Inteligência Artificial
NAC	Condição de Aplicação Negativa
DIAC	Regra de Aplicação Distinta Individual
UML	Unified Modeling Language
POS	Part of Speach
PLN	Processamento de Linguagem Natural
NER	Name Entity Recognition
DSL	Domain Specific Language
M2M	Model to Model
M2T	Model to Text
T2M	Text to Model
ES	Engenharia de Software
LHS	Left-Handside
RHS	Right-Handside

LISTA DE SÍMBOLOS

O(n)	Complexidade polinomial
O(n!)	Complexidade exponencial
$f \circ g$	Composição das funções f e g
f: $D \to I$	Função f de domínio D e imagem I
$A \subseteq B$	Conjunto A está contido em B
$a \in B$	a pertence a B
$G \hookrightarrow H$	G é subgrafo de H
ϕ	conjunto vazio

SUMÁRIO

1 INTRODUÇÃO

Manutenção de sistemas é uma atividade dispendiosa tanto para desenvolvedores quanto para usuários. Usualmente, no processo de desenvolvimento de sistemas existe demanda por pequenas modificações em produtos de software de forma concomitante ao desenvolvimento de novos projetos, com desafios e prazos próprios. Nesses casos, o cliente acaba sem opções além de aguardar por cronogramas longos e serviços caros. Diante dessa situação, evidencia-se que algum grau de autonomia na manutenção do sistema poderia trazer benefícios para todos os envolvidos. Ciente disso, essa dissertação propõe o uso de técnicas de transformação e síntese de programas com intuito de automatizar uma parte desse trabalho.

Mesmo reconhecendo que algumas atividades de manutenção em sistemas computacionais são bastante peculiares, podendo envolver extensa refatoração de código, mudanças em regras de negócio e modificação em dependências ao longo de diversos níveis de arquitetura, a hipótese desta dissertação é que: dado que o sistema seja representado com um nível de abstração adequado, alterações úteis podem ser realizadas por meio de transformações de grafo, possibilitando a um usuário decidir quando e onde aplicar essas transformações. A partir dessa ideia, o modelo MITRAS foi desenvolvido para trabalhar no escopo de manutenções adaptativas e perfectivas. Conforme definição do SWEBOK (BOURQUE; FAIRLEY et al., 2014) as manutenções adaptativas e perfectivas não implicam o não funcionamento do sistema, mas compreendem a ideia de melhorias ou adaptações a requisitos que se modificaram após a entrega.

Essa dissertação apresenta um novo modelo para transformação de software, delineando sua base teórica de transformações de grafo e mostrando a experimentação prática de um protótipo funcional aplicado em um projeto *open source*. Ambos os escopos, teórico e prático, foram considerados com o intuito de tornar o modelo coerente do ponto de vista lógico-formal e prático do ponto de vista da aplicação.

Outros trabalhos abordam situações semelhantes a essa, na revisão de literatura apresentada em (KAHANI et al., 2018), pode-se ter uma boa ideia de diversas ferramentas empregadas para transformações de modelos de software. Contudo, nossa proposta difere dessas ferramentas e de outros trabalhos (EHRIG et al., 2015a; KOCH; MANCINI; PARISI-PRESICCE, 2001; AMIRAT et al., 2012) principalmente por combinar transformações do tipo *Model-to-Model* e *Model-to-Text* simultaneamente e gerar uma interface adequada a usuários e não apenas a desenvolvedores.

Em uma situação mais tradicional, na qual apenas desenvolvedores podem fazer modificações no sistema, é necessário que o programador entenda a necessidade do usuário e traduza-a para código fonte. Com intuito de permitir que software possa ser manipulado por pessoas sem experiência em desenvolvimento, sistemas de manutenção automatizada necessitam reunir simultaneamente conhecimento de domínio e programação (BONDY; MURTY et al., 1976). Conhecimento de domínio fornece a semântica necessária para a comunicação com o usuário,

enquanto conhecimento de programação fornece capacidade de utilizar transformações para síntese de novas funcionalidades. No MITRAS, conhecimento de domínio e de programação são representados via ontologias. Essas ontologias correlacionam conceitos e entidades de aplicação, fornecendo um vocabulário semântico utilizado para mapear requisições feitas em linguagem natural para transformações de software.

Existem diversas formas para tratamento formal de transformações de software, podendo em alguns casos até mesmo se considerar transformações diretamente no nível do código fonte (EHRIG et al., 2015b), porém a abordagem desse trabalho envolve técnicas de transformação formal que operam em um nível mais abstrato, considerando um modelo do software implementado com Grafos. Para viabilizar essas transformações, o modelo conta com um processo de extração de modelo e de injeção de código fonte, responsáveis por atualizar o modelo de grafos e o código fonte, respectivamente.

O conhecimento de como transformar um software para incorporar novas funcionalidades é representado no MITRAS por regras de transformação de grafos, formalmente equivalentes a produções de grafos utilizadas em gramáticas de grafo (ROZENBERG, 1997; EHRIG; KORFF; LöWE, 1991; LöWE, 1993). A expressividade de regras de transformação, ou seja, a quantidade de modificações em um sistema que podem ser representadas via transformações de grafo, é dada pela granularidade do modelo escolhido. Modelos que possuem todos os detalhes do código fonte mapeados podem prover ao mecanismo de transformação a capacidade de representar qualquer edição de código fonte. Apesar disso, o projeto MITRAS trabalha com transformações em mais alto nível, que não envolvem edição de código fonte dentro de métodos. Dois tipos de manutenções de alto nível são tratadas pelo MITRAS, são elas:

(a) Adicionar funcionalidades: Alterações que envolvem adicionar novas entidades, atributos e processos no software ou mudanças de interface que podem ser incorporadas no modelo do software

(b) Manipulação de Interface de Usuário: Ocultar ou mudar a organização de entidades e atributos da camada de apresentação do sistema, diminuindo a carga cognitiva da interface de usuário, sem efetivamente remover a informação do sistema.

1.1 Definição do Problema

De forma geral, é fato intuitivamente reconhecido por profissionais de desenvolvimento que a manutenção é uma etapa na qual é investida uma parcela significativa do esforço em desenvolvimento de sistemas (PIGOSKI, 1996; ERLIKH, 2000; BUCHMANN; FRISCHBIER; PUTZ, 2011). Contudo, é comum que esta percepção seja associada a ideia que manutenção de sistema é apenas correção de problemas, o que não é o caso. De fato, em algumas situações, 80% do esforço empregado na fase de manutenção não é utilizado para correção de problemas (PIGOSKI, 1996).

Tabela 1 – Respostas do Questionário

Tipo/Esforço	Baixo(0% - 39%)	Médio(40% - 59%)	Alto(60% - 100%)
Corretiva	83.3%	0%	16.7%
Preventiva	83.4%	16.7%	0%
Perfectiva	25%	33.3%	41.7%
Adaptativa	83.3%	8.3%	8.3%

O modelo proposto nessa dissertação busca uma alternativa para mitigar o esforço empregado nas atividades de manutenção não-corretivas. Utilizando técnicas de transformação de grafos, é possível automatizar algumas atividades de manutenção, dando ao usuário final do sistema autonomia para aplicar as modificações onde ele julgar mais pertinente, dado que não sejam correções de bugs.

Com intuito de obter dados mais atualizados, validando a informação de que manutenções não corretivas são parte expressiva do esforço de desenvolvimento, foi elaborado um questionário entregue em empresas da região metropolitana de Porto Alegre (RS), questionando sobre a percepção do esforço gasto em manutenções. As perguntas propostas estão listadas a seguir:

1. Considerando o tempo total gasto com manutenção de sistemas, qual percentual desse tempo é gasto com manutenções corretivas? (um bug encontrado por usuário em uma funcionalidade já liberada)

2. Considerando o tempo total gasto com manutenção de sistemas, qual percentual desse tempo é gasto com manutenções preventivas? (um bug encontrado pela equipe de desenvolvimento em funcionalidade já liberada, que ainda não foi visto pelo cliente)

3. Considerando o tempo total gasto com manutenção de sistemas, qual percentual desse tempo é gasto com manutenções perfectivas? (novas features solicitadas pelos clientes)

4. Considerando o tempo total gasto com manutenção de sistemas, qual percentual desse tempo é gasto com manutenções adaptativas? (adaptações realizadas no software para ele se adequar a uma nova infraestrutura, hardware ou sistema)

Na Tabela 1, podemos identificar o resultado compilado da resposta das 12 empresas contatadas que se disponibilizaram a responder o questionário. De cada uma das empresas, um profissional de perfil senior ou um gestor respondeu. Para termos uma breve ideia do perfil dos profissionais que responderam a pesquisa em nome das empresas, em sua maioria trabalham com aplicações do mercado corporativo (75%), trabalham em uma empresa com mais de 50 desenvolvedores (50%) e possuem mais de 6 anos de experiência com desenvolvimento de sistemas (50%).

Esses resultados fornecem evidência de que manutenções perfectivas parecem demandar maior esforço do que as demais, com 75% das respostas classificando essa atividade com es-

força de desenvolvimento médio ou alto. Em contrapartida, outros tipos de manutenção requerem um esforço aparente menor, sendo que 83.4% das respostas classificaram esse tipo de atividade com baixo esforço de desenvolvimento.

O questionário também previa informações quanto ao tamanho (em termos de equipe de desenvolvimento) da empresa, o tamanho das empresas foi classificado como grande (50 desenvolvedores ou mais) e média/pequena (menos do que 50 desenvolvedores). Nossa amostragem consiste de 6 respostas contemplando empresas classificadas como média/pequena e 6 empresas classificadas como grandes. O cruzamento dos dados de tamanho das empresas com as respostas do questionário mostrou que existe uma diferença entre as respostas das empresas quando realizamos uma análise considerando separadamente esses dois subgrupos de informações, enquanto 50% das empresas grandes consideram manutenções perfectivas ou adaptativas como atividades que demandam alto esforço, esse número cai para 33.3% quando analisamos apenas empresas médias/pequenas. Devido a diferença no tamanho da força de trabalho, é possível inferir que provavelmente existem mais desenvolvedores engajados em atividades de manutenção adaptativas e perfectivas em empresas do que nos demais tipos de manutenção. Embora a amostragem seja pequena para realizar alguma afirmação com base estatística, ela confirma as evidências já apresentadas pela literatura.

Esses resultados corroboram a ideia de que manutenções não-corretivas de sistemas representam um esforço considerável no desenvolvimento de software, fomentando a importância da pesquisa para encontrar novas ferramentas e métodos que otimizem esse tipo de trabalho. É reconhecido que a amostragem utilizada na pesquisa é limitada em termos de tamanho, e pesquisas mais extensivas devem ser realizadas para caracterizar melhor essa situação. Contudo, acredita-se que a amostra é representativa dentro de sua área de estudo, visto que os resultados estão consonantes com dados bibliográficos (ver seção 2.1).

Dessa forma, existe motivação para trabalhar com sistemas que possam automatizar atividades de desenvolvimento, diminuindo custos e prazos para todos os envolvidos.

Certamente que esse tipo de sistema ainda não tem um uso difundido nas empresas de desenvolvimento de software, assim essa dissertação se propõe a explorar como questão geral de pesquisa se: Um sistema de transformações pode realizar alterações em um software arbitrário com resultados semelhantes a um desenvolvedor a partir de interações em linguagem natural?

Essa pergunta é relevante pois encontramos na literatura (Ab. Rahim; WHITTLE, 2015; JILANI; USMAN; HALIM, 2010; KAHANI et al., 2018; SENDALL; SENDALL; KÜSTER, 2004) evidências que apontam para o benefício de ferramentas de transformação durante o processo de desenvolvimento. A partir da viabilidade de acionar um conjunto de ferramentas de transformação por meio de interações em linguagem natural, a proposta do MITRAS de permitir que usuários sem conhecimento técnico de programação possam se valer dessas mesmas tecnologias torna-se possível.

Para responder essa questão de pesquisa foi necessário criar um protótipo implementando as funcionalidades de processamento de linguagem natural e todas as etapas de transformação

necessárias para sintetizar as alterações no sistema. Esse protótipo foi então colocado em testes para coletar evidências sobre o uso do MITRAS comparando código gerado e tempo empregado com alterações realizadas por desenvolvedores, a metodologia pode ser consultada de forma mais detalhada em 1.3.

1.2 Objetivo

Da mesma forma que não se espera que um usuário saiba programar para poder utilizar um sistema qualquer (salvo em casos específicos, mas aqui estamos nos referindo em sistemas de usuário final), o uso do MITRAS não deve ter como premissa conhecimento técnico em computação, mas apenas conhecimento do domínio de aplicação sobre o qual o sistema foi construído.

O objetivo desse trabalho é propor um modelo formal para transformações de software que permita criação de novas funcionalidades sem conhecimento técnico de programação. Dessa forma, usuários sem familiaridade com lógica de programação e ferramentas de desenvolvimento ganham uma alternativa para solicitar mudanças que não exigem modificação direta em algoritmos do sistema. Com intuito de atingir esse objetivo geral, os objetivos específicos abaixo foram definidos:

(a) Criar um modelo de transformações de software formal, baseado em gramáticas de grafo;

(b) Realizar experimentações com o modelo proposto em um software *open source*;

(c) Aplicar transformações a partir de solicitações em linguagem natural; e

(d) Comparar alterações realizadas por programadores com alterações sintetizadas pelo modelo.

1.3 Metodologia

Essa seção vai detalhar a metodologia utilizada no desenvolvimento da dissertação, contemplando procedimentos de pesquisa bibliográfica e experimentação.

A primeira etapa desse trabalho foi buscar bibliografia referente ao estado da arte de Programação Automática, Síntese de Programas e geração de código. Essas disciplinas possuem objetivos similares a essa dissertação e suas pesquisas aparecem normalmente vinculadas a área de Engenharia de Software Automatizada. O resultado dessa pesquisa pode ser encontrado no capítulo 3 de trabalhos relacionados.

A partir da compreensão dos trabalhos relacionados, percebeu-se que **Grafos** são uma opção sólida para representação de sistemas computacionais. Dessa forma, optou-se por selecionar um formalismo de transformações de grafos que tivesse resultados concretos e um corpo de pesquisa ativo. Definiu-se a utilização da Abordagem Algébrica com *single-pushout* para

transformações de grafo. Uma explicação mais detalhada dos mecanismos de transformações de grafo é feita na fundamentação teórica, no capítulo 2, onde também pode ser encontrado conteúdo para o leitor se familiarizar com Manutenções de Software, Ontologias e Grafos.

Tendo a carga teórica necessária, fez-se a formalização do modelo. Para tanto, conceitos como a categoria de grafos G, grafo, morfismo, derivação e aplicação de produção foram retomados e formalizados, de modo que todo o aparato teórico necessário para operacionalizar o protótipo com capacidade para transformações seja conhecido e definido.

Como os objetivos implicam a aplicação do modelo em um sistema real, existe também a necessidade de encontrar um sistema *open source* para viabilizar o experimento. Esse sistema é então submetido a um processo de extração do modelo de grafos considerando mineração de repositórios de código fonte e execução assistida do sistema. Essa mineração utiliza basicamente algoritmos equivalentes a etapa de parse de um compilador, pois constrói toda a estrutura sintática do programa dando atenção especial a suas dependências. A principal diferença desse processo de extração de modelo para uma compilação é a saída, que passa a ser um modelo de grafos e não um código executável.

De posse de uma base de código fonte e do modelo, iniciou-se a implementação do protótipo, estudando a aplicação de quatro tipos de transformações. Essas transformações dão conta de adicionar campos persistentes, ocultar informações na interface de usuário, reorganizar a interface e criar novas funcionalidades a partir de padrões de projeto. Essas transformações foram projetadas e executadas considerando a base de código fonte escolhida e os dados sobre tempo de execução e complexidade foram devidamente registrados.

Com os resultados dos experimentos, é proposta uma discussão da aplicabilidade do modelo em sistemas de diversas arquiteturas e paradigmas nas considerações finais dessa dissertação.

A segunda etapa de experimentos consiste em dois grupos de pessoas, um formado por desenvolvedores de perfil júnior e outro por usuários. Aos indivíduos de ambos os grupos foi solicitado que realizassem alterações no sistema, porém, enquanto o grupo de desenvolvedores recebeu uma contextualização sobre as mudanças e acesso ao código fonte, o grupo de usuários recebeu acesso ao MITRAS. Dessa forma são comparados qualitativamente e quantitativamente os resultados dos dois grupos com intuito de avaliar a solução criada pelo MITRAS em comparação com a solução criada pelos desenvolvedores para as mesmas solicitações. O principal resultado encontrado nessa etapa é que em experimento controlado, tanto desenvolvedores com acesso a código fonte quanto usuários com acesso ao MITRAS conseguiram realizar 92% das alterações propostas, porém os usuários levaram em média 8 minutos enquanto os desenvolvedores levaram em média 26 minutos.

2 FUNDAMENTAÇÃO TEÓRICA E TECNOLÓGICA

Neste capitulo serão apresentados os conceitos teóricos e ferramentas utilizadas como base para o desenvolvimento do modelo MITRAS e de sua implementação prática. A primeira seção apresenta o conceito de manutenção de software e delimita para quais tipos de manutenção o modelo proposto pode auxiliar, após isso são apresentadas seções sobre processamento de linguagem natural, a forma escolhida para a interação com o modelo e ontologias que auxiliam no processo de processamento de linguagem natural. As seções subsequentes trabalham com ideias ligadas a etapa de transformação de grafos no sistema, são realizadas definições da teoria de grafos culminando no estudo de transformações de grafo.

2.1 Manutenção de Software

Na Engenharia de Software (ES), manutenção trata sobre o conjunto de atividades de desenvolvimento realizadas após a entrega do produto, contemplando melhorias, correções de problema bem como todas as atividades necessárias para manter o sistema em uso. A manutenção é um dos processos fundamentais no ciclo de vida do software (ISO/IEC 12207 - Software life cycle processes, 2008).

Conforme definição do SWEBOOK V3 (BOURQUE; FAIRLEY et al., 2014), é possível dividir os tipos de manutenção, desde a implementação de novos requisitos solicitados pelo cliente, passando pela correção de não conformidades e chegando em melhorias de requisitos não funcionais do sistema, conforme classificação a seguir.

- manutenção corretiva: modificações para corrigir erros ou falhas operacionais;
- manutenção adaptativa: modificações para que o software possa continuar a ser usado no ambiente de hardware e software;
- manutenção perfectiva: modificações para implementar melhorias, fazer customizações ou adicionar *features* para os usuários;
- manutenção preventiva: modificação para corrigir falhas latentes, antes que se tornem falhas operacionais.

Existem outras formas de classificar manutenção quanto sua natureza, a norma ISO/IEC 14764 (ISO/IEC 14764 - Software Maintenance, 2006) é bastante consonante ao SWEBOOK, porém vai além e define processos para tratamento de cada tipo de manutenção apresentado acima. Contudo, é necessário salientar que a padronização proposta pela norma não resolve todas as dificuldades inerentes a manutenção de sistema, servindo apenas como um *guideline*.

Dois tipos de manutenção merecem um enfoque especial no contexto dessa dissertação, manutenções adaptativas e perfectivas. Em ambos casos, a necessidade de alteração no sistema

não está associada necessariamente a um defeito, e sim a uma característica nova a qual o sistema não é aderente. Um exemplo de manutenção adaptativa é a simplificação de uma interface de usuário, com intuito de se adaptar melhor a ambiente *mobile*, enquanto uma manutenção perfectiva poderia ser uma solicitação de novas informações em um determinado cadastro.

É relevante se preocupar com a manutenibilidade do software desde seu projeto. Sistemas tipicamente estão inseridos em ambientes cuja mudança de requisitos é constante, então um dos fatores que influencia diretamente no sucesso de uma aplicação computacional é a facilidade que o sistema tem a se adaptar a mudanças (PRESSMAN, 2005).

Ao contrário daquilo que acontece com sistemas mecânicos, sistemas de informação não possuem peças que degradam fisicamente, assim manutenções de software visam evitar a deterioração do sistema frente a seus requisitos. Na prática, conforme o tempo passa e o ambiente se modifica, o software passa a ser cada vez menos aderente ao ambiente para o qual foi projetado (PADUELLI, 2007).

A necessidade prática por modelos e ferramentas que melhorem a forma com a qual trabalhamos com manutenção de sistemas computacionais é corroborada também pelo aspecto financeiro. Em desenvolvimento de sistemas, tipicamente as despesas com atividades de manutenção excedem as etapas de engenharia e criação. Algumas organizações gastam 80% do orçamento do sistema com manutenção (PIGOSKI, 1996), mas esses valores podem ultrapassar os 90% (ERLIKH, 2000). Em alguns casos, mesmo em empresas cuja atividade final não é o desenvolvimento de sistemas, o custo com manutenção de sistemas pode passar de 25% de todo orçamento com TI (BUCHMANN; FRISCHBIER; PUTZ, 2011).

De fato, a necessidade de pesquisa e ferramentas de apoio ao processo de manutenção de sistemas ainda é presente. Nos últimos quarenta anos, o volume de pesquisa nesse tópico vem aumentando, técnicas cada vez mais rebuscadas e análises de um volume maior de dados é feita. Apesar disso, ainda não existe um modelo óbvio ou uma ferramenta sólida para apoio da tratativa de manutenções (LENARDUZZI; SILLITTI; TAIBI, 2017).

2.2 Processamento de Linguagem Natural

O que uma linguagem natural possui de idiossincrasias, uma linguagem formal possui de regularidades. Embora ambas formas de linguagens sejam criadas pelo ser humano, ao passo que a linguagem natural surgiu de forma orgânica no processo evolutivo, linguagens formais foram minuciosamente projetadas. Dessa forma, linguagens naturais são utilizadas para comunicação entre seres humanos e linguagens formais são modelos matemáticos que possibilitam especificar e reconhecer uma linguagem específica.

No estudo de linguagens formais, precisamos definir os conceitos de símbolo, alfabeto, cadeia e linguagem. Símbolos são os elementos básicos da estrutura que compõe uma cadeia e, por consequência, uma linguagem. Esses símbolos normalmente pertencem a um conjunto finito de símbolos da linguagem, ao qual damos o nome de alfabeto. Uma cadeia é formada

pela operação de concatenação (justaposição) de uma quantidade arbitrária de símbolos. Por fim, uma linguagem é um conjunto finito ou infinito de cadeias cujo comprimento é finito e sua formação obedece a um dado alfabeto (RAMOS, 2008).

Linguagens formais que despertam mais interesse são constituídas por um conjunto infinito de cadeias, desse modo, a ideia mais direta para a representação da linguagem, uma listagem das cadeias válidas, não é viável. Nesses casos, nos quais não é possível conhecer de antemão todas as cadeias válidas, é necessário trabalhar com gramáticas e reconhecedores.

Uma gramática é um instrumento formal de geração de cadeias, de modo que todas as cadeias válidas em uma determinada linguagem formal podem ser geradas por uma gramática específica. Em contrapartida existe o reconhecedor, que é um algoritmo capaz de ler uma cadeia de símbolos e identificar se essa cadeia faz parte ou não de determinada linguagem. Ambas as ferramentas, gramáticas e reconhecedores, são utilizadas em linguagens cujo conjunto de cadeias válidas seja finito ou infinito.

É importante frisar que gramáticas e reconhecedores são igualmente expressivos, ou seja, se é possível projetar uma gramática para gerar uma linguagem existe um reconhecedor que valida somente as cadeias válidas dessa linguagem (AHO; SETHI; ULLMAN, 2007).

Para compreender completamente uma linguagem, deve-se atentar tanto para a forma de escrita e disposição dos símbolos (sintaxe) quanto ao significado dessas construções (semântica). Em uma linguagem formal, ambos aspectos se complementam, sendo o significado das construções linguísticas desenvolvido em estudos de semântica formal.

Gramáticas de linguagens formais trabalham com uma série de regras para geração das cadeias válidas em uma linguagem. Basicamente as regras utilizadas constituem um sistema formal de substituição de símbolos de acordo com determinadas símbolos terminais ou não-terminais.

Outra forma de nomear as substituições em uma gramática é a noção de derivação direta de uma cadeia. É comum definir linguagens especificas para denotar o uso de gramáticas, entre as formas de representação mais comuns estão as notações algébricas, notação de Backus-Naur (BNF) e diagramas de sintaxe (também conhecidos como Diagramas Ferroviários), todas as formas de representação listadas são equivalentes em termos de expressividade.

Um tipo de gramática comumente utilizado na definição de linguagens de programação é a gramática livre de contexto (HOPCROFT, 2008) Uma gramática desse tipo é composta de quatro componentes principais: Um conjunto de símbolos terminais, um conjunto de símbolos não-terminais (ou variáveis sintáticas), um conjunto de produções (ou derivações) que especificam uma regra de substituição para um símbolo não terminal e um símbolo gerador inicial.

Essa estrutura geradora de uma gramática livre de contexto gera, de forma semelhante a linguagem natural, uma árvore de parse. Na qual ficam explícitos os símbolos iniciais, os símbolos não-terminais e as produções aplicadas para obtenção dos símbolos terminais. Em casos de uso reais, como no caso de compiladores, dificilmente toda a árvore de parse é montada, apenas pequenos trechos dela são instanciados para resolver partes do problema (AHO; SETHI; ULL-

MAN, 2007).

Compiladores são talvez os programas reconhecedores mais conhecidos a se beneficiarem de definições formais de linguagens de programação, de fato é interessante atentar que esse tipo de programa tipicamente cria uma representação de código intermediária antes de realizar a tradução de código fonte para código de máquina. Essa capacidade será explorada no contexto dessa dissertação, pois as ferramentas de compilação de linguagens formais serão utilizadas para a criação de uma representação intermediária no formato de grafos. A teoria aplicada se mantém a mesma, o que muda nesse caso é que ao invés de trabalhar com uma representação próxima da linguagem de máquina é trabalhado com uma representação de grafos.

Uma etapa importante do processo de compilação de um código é a análise semântica, na qual, a partir da árvore de parse (ou da árvore de sintaxe abstrata) e das regras semânticas da linguagem, o código é avaliado em termos de consistência semântica. É nessa etapa que avaliações importantes, como é o caso da tipagem de dados, são executadas.

Para fazer o processo de *parsing* de uma linguagem formal, emergem técnicas semelhantes as utilizadas para linguagem natural. Dentre elas a árvore de parse, que identifica sintaticamente todos os elementos de uma cadeia conforme seu papel. Por se tratar de uma linguagem formal, esses procedimentos são menos ambíguos, visto que as linguagens são projetadas para facilitar o processo de interpretação e interpretações erradas podem ser inaceitáveis dentro do escopo de aplicação.

A linguagem está fortemente ligada as nossas atividades do dia a dia, é necessário apelar para linguagem para adquirir ou difundir informação. Devido a isso, é natural que o corpo de informação disponível em linguagem natural esteja em constante crescimento, de modo que o interesse por técnicas que possam acessar e extrair informação desse montante de dados também aumente. Processamento de Linguagem Natural (PLN) é a área de pesquisa que estuda como utilizar essa quantidade abundante de informação que está disponível em linguagem natural (RUSSELL et al., 2013).

Linguagens de programação são ditas formais, pois possuem um modelo bem definido. Uma linguagem pode ser definida por um conjunto de strings, porém, dado que esse conjunto normalmente é infinito, sua definição não é realizada por extensão e sim por um conjunto de regras de produção, que damos o nome de gramática. Outra característica de linguagens de programação, sob o prisma de processamento de linguagens, é que a semântica desse tipo de linguagem também é bem definida, dado uma string que pertença a gramática dessa linguagem podemos atribuir um significado inequívoco a mesma.

Contudo, linguagens naturais não são tão bem comportadas quanto as formais. Linguagens como português ou inglês possuem frases nas quais não existe um consenso no que diz respeito ao seu significado, dessa forma não é possível definir um conjunto (mesmo que infinito) de frases pertencentes a uma linguagem natural, nesses casos acaba sendo realizada uma abordagem probabilística (RUSSELL et al., 2013).

Na esfera de PLN, podemos classificar os tipos de problema enfrentados quanto sua natu-

reza. A seguir são listados os tipos de problemas representativos em processamento de linguagem natural.

- Classificação/Categorização: São os casos nos quais um texto deve receber uma classificação dentre algumas classes previamente definidas, um exemplo é o problema de classificação de e-mails como spam ou ham;

- Recuperação de Informação: Ocorre quando existe um corpo de documentos a ser pesquisado com intuito de encontrar o documento mais relevante a partir de uma string enviada ao sistema, os buscadores da web como o Google trabalham com esse tipo de questão;

- Extração de Informação: Neste caso, existe um texto ou um corpus de texto que precisam ser lidos de forma a extrair informações relevantes;

- Comunicação: Diz respeito a troca intencional de sinais entre duas entidades de modo a transmitir uma determinada mensagem, sistemas de chatbot se enquadram nessa categoria. Pode ser considerado o problema mais complexo dentre os anteriores.

Mesmo reconhecendo a diversidade de problemas ao se trabalhar com processamento de linguagem natural, existem algumas noções que permeiam uma boa parte desses problemas. A noção de categoria léxica diz respeito a taxonomia das palavras, ou a classe gramatical das mesmas (verbo, adjetivo, ...), que podem ser trabalhadas em conjunto para se chegar as categorias sintáticas (Sujeito, Predicado, ...). A partir da análise lexical e sintática, é possível representar frases em estruturas de árvores, sendo que cada nível da árvore representa um nível da análise.

Um aspecto digno de nota no escopo dessa dissertação é a intenção por trás de uma mensagem enviada em linguagem natural, em casos práticos cabe ao receptor da mensagem intuir a intenção do emissor. A essa intenção, que define se a frase é uma pergunta, uma ordem, entre outros, dá-se o nome de atos da fala (SEARLE, 2002; AUSTIN, 1990).

Conforme já foi mencionado, as restrições que tornam linguagens formais mais determinísticas em sua análise não se aplicam a linguagens naturais. De fato, existem diversas situações que são potenciais problemas em processamento de linguagem natural, ambiguidade léxica e sintática são exemplos nos quais uma palavra ou frase pode ter uma função diferente dado o contexto em que se encontra. Esse tipo de situação pode ocasionar também uma ambiguidade semântica.

Outro desafio ao avaliar linguagem natural é a diferença entre significado literal e figurativo. Nesse caso, figuras de linguagem como (mas não limitadas a) metonímias e metáforas podem dificultar ainda mais a avaliação automatizada de frases tanto em relação a sintaxe quanto a semântica. Nesses casos uma aplicação sensível ao contexto pode melhorar os resultados, mas continua não sendo efetiva em todas as situações (RUSSELL et al., 2013).

Genericamente, ao processo de avaliação de uma estrutura textual quanto a suas características léxicas, sintáticas e semânticas, chamaremos de *parse* de uma linguagem. As subseções

seguintes trazem de forma resumida as principais técnicas e ferramentas empregadas no problema de *parse* de uma linguagem natural.

2.2.1 Part of Speech Tagger

Ao realizar uma análise gramatical de uma frase, uma das observações que é possível fazer em relação as palavras e como elas estão organizadas no texto é referente a sua classe gramatical.

Um sistema capaz de receber como entrada um texto em determinado idioma e associar as palavras sua classe gramatical, damos o nome de *Part of Speech Tagger* (POS Tagger). Para realizar esse processo de *tagging*, normalmente é definido um conjunto de tags padronizado, facilitando a interpretação e o uso desses sistemas. Dentre as opções disponíveis, se destaca o conjunto de tags do *Penn Treebank Project* (SANTORINI, 1990).

Diversos algoritmos de aprendizado de máquina são empregados para criação de *taggers* mais eficazes, entre as abordagens com melhores resultados podem ser destacados o uso de Modelos Ocultos de Markov e modelos de máxima entropia, os resultados obtidos com essa abordagem chegam a acertar mais de 96% dos casos (para o idioma inglês) sendo que sua principal dificuldade é trabalhar com palavras desconhecidas (TOUTANOVA; MANNING, 2000).

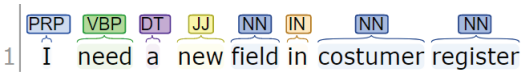

Figura 1 – Frase processada pelo POS Tagger

A Figura 1 exibe o resultado de uma frase após o processamento de um POS Tagger. Para cada uma das palavras é atribuída uma *tag* que representa uma classe gramatical. Apesar da classe gramatical das palavras ser importante para análise do significado e da intenção da frase, apenas um POS Tagger não consegue identificar relações entre palavras, para isso outras técnicas são necessárias.

2.2.2 Name Entity Recognition

Nomes próprios e números escritos em formato não decimal são comumente encontrados durante processamento de linguagem natural. A técnica *Name Entity Recognition (NER)*, é usada para reconhecimento de nomes e entidades, identificando nomes próprios e numerais. Esse tipo de distinção ocorre tanto pelo contexto da frase quanto a partir de vocabulários pré-definidos de termos e conceitos.

Observando a Imagem 2, é possível perceber que tanto o nome pessoal quanto a palavra *first* foram identificados pelo algoritmo de *Name Entity Recognition* e tiveram seus significados expostos. Dado um problema de PLN arbitrário, contar com a identificação de entidades conhecidas pode auxiliar no processamento do texto.

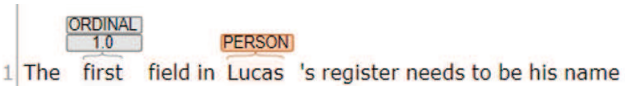

Figura 2 – Frase processada por Name Entity Recognition

Uma abordagem para trabalhar com NER é a utilização de algoritmos probabilísticos, sendo que ao adicionar informações não-locais é possível ter taxas de acerto melhores para esse tipo de identificação(FINKEL; GRENAGER; MANNING, 2005).

Mesmo considerando que é possível inferir palavras que indicam um nome próprio ou um numeral, uma das formas de trabalhar com *Name Entity Recognition* envolve o uso de ontologias, de modo a criar uma classificação dos termos conhecidos para posterior consulta no A-box da ontologia. Munindo-se desse artifício, um sistema de reconhecimento de entidades pode ser melhorado a partir da inserção de novos termos na ontologia.

2.2.3 Árvore de Parse

A árvore de parse é uma forma de representação de uma frase, de modo a evidenciar classes gramaticais de palavras e funções de orações dentro da frase. Sua estrutura em árvore favorece a navegação entre os nodos e a criação de uma análise sintática bem definida e com diversos níveis de detalhamento. Nessa estrutura, a raiz da árvore é a frase inteira, a qual é ramificada em orações e partes de frase, até chegar nas palavras individuais que são representadas como folhas da árvore.

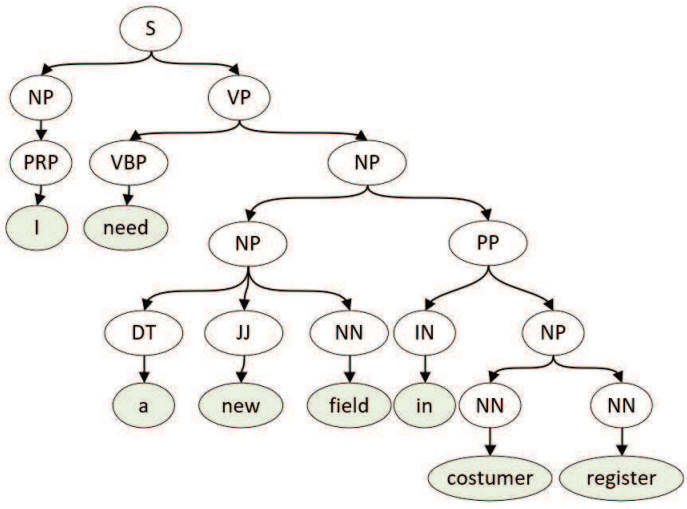

Figura 3 – Exemplo de árvore de parse

Essa estrutura de dados não é utilizada apenas para parse de linguagens naturais, de fato em linguagens formais, uma das técnicas utilizadas por compiladores para verificação da validade sintática de um dado código envolve a construção parcial de uma árvore de parse(AHO; SETHI; ULLMAN, 2007).

A árvore de parse da frase *"I need a new field in costumer register"* está ilustrada na Figura 3. Nessa árvore, podemos perceber que as palavras ficam destacadas nas folhas da árvore, quanto os nodos intermediários identificam classes gramaticais das palavras, como adjetivos e verbos. É interessante perceber que as folhas não ficam necessariamente no mesmo nível, isso depende da estrutura da frase.

2.2.4 Grafo de Dependências

Quando trabalhamos com processamento de linguagem natural, um dos desafios é a identificação de dependências entre uma ou mais palavras de uma dada frase. Uma das formas de criar uma representação computável dessa estrutura é utilizando grafos.

Consideremos que $S = w_1, w_2, ..., w_n$ é uma frase onde cada w_i representa uma palavra dessa frase, $G = (V, A)$ é o grafo de dependências dessa frase sendo que $V = (w_1, w_2, ..., w_n)$ são os vértices e $A \subseteq V \times V$ é um conjunto de arestas que representa as dependências entre as palavras. A Figura 4 mostra um grafo de dependências gerado a partir de uma frase arbitrária.

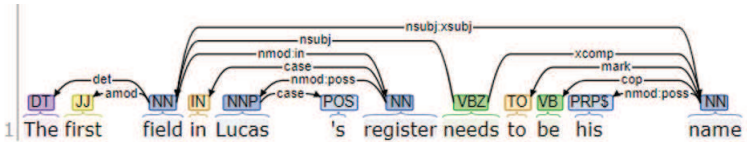

Figura 4 – Exemplo de grafo de dependências

Como é possível identificar na Figura 4, o grafo em questão é dirigido. A direção das arestas carrega significado para as relações, um exemplo é a relação de verbo e objeto, que nos permite identificar qual dos extremos da aresta é o verbo e qual é o objeto.

Quando falamos sobre a detecção de dependências entre palavras estamos trabalhando com pares de palavras, de modo que uma delas possui uma função em relação a outra. Os *parsers* mais tradicionais que trabalham nessa questão precisam da entrada de um número considerável de *features* para chegar a uma boa acurácia, de fato, 99% do tempo de processamento de um parser desses pode ser gasto apenas em extração de *features*(CHEN; MANNING, 2014). Contudo, existem modelos baseados em redes neurais que trabalham com um número de entradas menor e acabam minimizando esse processamento.

A representação das dependências de uma frase não é uma tarefa trivial, diversas abordagens para tentar definir um conjunto de dependências universal, que pudessem ser adotados em qualquer idioma foram realizadas. O grupo de pesquisa em processamento de linguagem natural de Stanford foi um dos primeiros a tentar criar uma padronização (2005, 2009 e 2013), seguidos de empresas privadas como a Google (2007 e 2011) até que em 2014 foi definido o *Universal Dependency Treebank* que conta com 100 treebanks de texto classificado em mais de 60 idiomas (DE MARNEFFE et al., 2014; NIVRE et al., 2016).

2.2.5 Parser de Stanford

Dentre as ferramentas disponíveis para parse de linguagem natural em diversos idiomas, destaca-se o Parser de Stanford (CHEN; MANNING, 2014). Ele conta com um pipeline de processamento capaz de realizar análise sintática e semântica de frases, contando com POS Tagger, Árvore de Parse, Grafo de Dependências, Name Entity Recognition, análise de sentimentos entre outras.

Desde 2010 o Parser de Stanford se tornou um software livre, sendo utilizado em diversas pesquisas acadêmicas sobre PLN bem como no meio empresarial. Um dos motivos ao qual é atribuído o destaque dessa ferramenta entre outras semelhantes é sua facilidade de uso, visto que não são necessários conhecimentos específicos de linguagem para utilizar o parser(MANNING et al., 2014).

Um diferencial desse parser, é a implementação do conjunto de dependências universais, que visa unificar o conjunto de tags e classificações utilizadas em palavras de forma independente do idioma. O que facilita que sistemas desenvolvidos com base em dependências universais possam ser reutilizados.

Existem diversas formas para integração desse parser a uma aplicação, entre elas é possível utilizar diretamente a API Java que está disponível ou trabalhar com um servidor web como parser e encaminhar requisições do tipo REST.

O parser de Stanford trabalha também com modelos probabilísticos e neurais previamente treinados, cabendo ao cliente da aplicação escolher qual dos modelos disponíveis é o mais adequado a sua aplicação, podendo inclusive treinar um modelo customizado caso seja necessário. Essas características tornam o parser de Stanford uma opção flexível e simples de utilizar.

2.3 Ontologias

A representação do conhecimento é uma área da Inteligência Artificial (IA) que estuda a manipulação de conhecimentos a partir de estruturas armazenadas em sistemas computacionais (SOWA et al., 2000). Existem diversas técnicas para abordar esse tema, dentre as possibilidades, o uso de ontologias vem se consagrando como uma opção sólida. De forma resumida, ontologias trabalham com representação simbólica baseada em Lógicas de Descrição (que em última instância são subsets do cálculo de predicados da lógica de primeira ordem).

O termo Ontologia diz respeito a uma área do conhecimento tradicionalmente vinculada a filosofia. Apesar de não cunhar o termo, é possível encontrar definições dessa disciplina em escritos de Aristóteles (Metafísica IV). Contudo, essa visão mais tradicional não será o enfoque dessa dissertação, estamos interessados aqui no uso de ontologias como ferramentas de representação de conhecimento em sistemas computacionais.

Dentro da esfera da representação do conhecimento, uma ontologia pode ser compreendida como uma teoria lógica que representa a semântica de entidades da realidade considerando suas

regras restritivas (GIARETTA; GUARINO, 1995). Dessa forma as ontologias podem criar e organizar conceitos da realidade bem como seus relacionamentos semânticos.

Extrair a semântica de determinados aspectos da realidade tende a ser uma atividade informal e embasada em experiências prévias. Apesar disso, o produto desse trabalho é um conhecimento formalizado que pode ser utilizado por sistemas computacionais nas mais diversas tarefas (VAN HARMELEN; LIFSCHITZ; PORTER, 2008).

Ao mesmo tempo que ontologias podem ser utilizadas para mapear conceitos abstratos como tempo, é possível empregar ontologias para domínios de conhecimento bastante específicos, como por exemplo na medicina. Essa variedade de aplicações para ontologias ilustra a necessidade de criar uma classificação das ontologias quanto sua generalidade.

Ontologias mais genéricas, que ilustram conceitos que não variam de domínio para domínio são classificadas como ontologias de topo, o segundo nível se divide em ontologias de domínio e de tarefas que buscam formalizar conceitos de domínios específicos como por exemplo veículos ou tarefas como por exemplo executar manutenção, ontologias de aplicação por outro lado possuem elementos de domínio e de tarefa e podem mapear um determinada função caracterizada por elementos de domínio e tarefas.

Além de formalizar o conhecimento, a engenharia de conhecimento aplicada a ontologias se preocupa com o reuso dessas informações, de modo que quando surge a necessidade de criar uma ontologia de domínio é possível reusar conceitos já formalizados em outras ontologias.

A possibilidade de reuso de ontologias surge da necessidade de mapear um novo domínio dado que existe sobreposição de conceitos com outras ontologias disponíveis, nesse caso o projetista da ontologia pode se preocupar em mapear esse ponto de intersecção e pode consumir todo o conhecimento mapeado na ontologia reutilizada. Essa capacidade fomenta a criação de ontologias mais abrangentes.

Na prática, a IA utiliza-se de ontologias para mapeamento de informações e, para tanto, necessita de uma linguagem bem definida, que se baseia usualmente em lógicas declarativas formais. Uma linguagem em evidência para projeto de ontologias computacionais é o padrão OWL, que nada mais é do que uma fração da lógica de primeira ordem com algumas propriedades que facilitam a manipulação do conhecimento (MUNN; SMITH, 2008).

Cabe ressaltar que a manipulação de teorias lógicas em ambientes computacionais não é um processo trivial, uma abordagem ingênua para trabalhar com uma ontologia OWL pode possibilitar a manipulação de estruturas de dados complexas, mas carece da capacidade de explicar a semântica por trás das classes, objetos e relacionamentos projetados. É somente quando se leva o caráter lógico da linguagem OWL que passa a ser possível explorar toda a sua expressividade.

O desenvolvimento de software pode se beneficiar em diversos aspectos ao incorporar ontologias. Porém, embora seja reconhecido que durante o projeto de sistemas de informação a representação de conhecimento ontológica pode ser útil (GUARINO, 1997), nessa pesquisa as ontologias serão utilizadas pelo MITRAS no processamento de linguagem natural, durante a

execução do sistema e não durante o projeto.

Sistemas de informação habitualmente são empregados para tratar problemas em ambientes de domínio específico de conhecimento. Ao incorporar uma ontologia, essas aplicações podem se valer da semântica do domínio para executar suas funções de forma mais próxima ao usuário final. Ontologias aplicadas em tempo de execução podem tanto ser um componente interno do sistema ou uma aplicação externa consultada via API para realização de atividades específicas.

Mapear funcionalidades de software em ontologias pode trazer vantagens ao usuário, como sistemas de busca semânticos ou também pode abrir uma gama diferente de aplicações tecnológicas. De forma genérica é esperado que os sistemas computacionais se adaptem a forma de pensar dos usuários e não vice-versa, para conseguir computar esse tipo de informação os sistemas necessitam de formalismos computáveis e flexíveis (SOWA, 1976).

Mesmo considerando a constante evolução tecnológica, trabalhar com a representação de conhecimento de domínio aliada a aspectos da implementação do sistema continua sendo um desafio. O ambiente de desenvolvimento conta com diversas metodologias e artefatos bastante heterogêneos, dificultando o processo de criação de uma base de conhecimento consolidada para realizar inferências a partir desse conhecimento (HOCHGESCHWENDER et al., 2016).

2.4 Grafos

Diversas situações podem ser diagramadas considerando dois objetos simples: pontos e linhas de conexão entre pontos. Tráfego aéreo, relação de amizade entre pessoas, redes de computadores e diagramas UML são apenas alguns exemplos de problemas que podem ser representados utilizando grafos, nome que se dá a essa abstração topológica (BONDY; MURTY et al., 1976).

A expressividade dos grafos é útil para tratar de problemas com informações diversas. No âmbito da arquitetura de sistemas, variados artefatos compõe um software: Código fonte em diversas linguagens, arquivos de configuração e definições de interface gráfica são exemplos que possuem suas próprias idiossincrasias. Dito isso, é preciso contar com um formalismo flexível caso se queira trabalhar com esse tipo de informação em um modelo consolidado.

A palavra grafo é um neologismo da palavra em inglês *graph*, que indica a capacidade desse tipo de estrutura ser representado graficamente, de modo a facilitar uma noção intuitiva de suas estruturas e propriedades. Um fator importante é que, não existe apenas uma maneira de desenhar um grafo, dado vértices e arestas devidamente definidos e ligados, a disposição deles em um plano não é necessariamente relevante.

Grafos são estruturas estudadas pela matemática discreta. Podemos definir informalmente um grafo como um conjunto não vazio de vértices (nodos) e um conjunto de arestas (arcos) que conectam dois vértices (GERSTING, 2014). Além dessa construção básica, é possível incorporar mais informação a esses objetos gerando uma rotulação com informações arbitrárias que pode ser associada tanto a arcos quanto a vértices do grafo.

Sempre que os arcos de um dado grafo possuem uma direção bem definida dizemos que ele é um grafo dirigido ou digrafo (GERSTING, 2014). Incorporar essa regra ao grafo pode parecer uma restrição, contudo a direção de um arco passa a incorporar mais informações ao modelo. Um exemplo de situação na qual a direção de uma aresta pode ser útil é na modelagem de um relacionamento de herança em UML, a partir da direção da relação definimos quem é a superclasse e quem é a subclasse.

No contexto dessa dissertação, estamos interessados em trabalhar com grafos dirigidos, finitos, não necessariamente acíclicos e com rotulação em vértices e arestas. Conforme a formalização do modelo for dada, substitui-se o conceito de grafos rotulados por grafos tipados, nesse caso mantém-se a equivalência entre ambas abordagens dado que a rotulação faz parte de um conjunto finito que define os tipos de arestas e vértices.

Questões que naturalmente se adaptam a representação de grafos normalmente se relacionam a verificação se dois pontos estão conectados. Na teoria de grafos esse problema é chamado de atingibilidade e diz respeito a existência de um caminho entre dois vértices passando por uma ou mais arestas. De fato, algoritmos para tratamento de problemas como a atingibilidade envolvem uma técnica de busca em grafos. Buscar em um grafo significa percorrer os arcos de forma sistematizada, partindo de um nodo inicial para chegar em um nodo final determinado, esse tipo de pesquisa pode descobrir muito sobre a estrutura de um grafo (CORMEN, 2009).

Dadas as considerações acima, estamos aptos a realizar a definição formal de grafos rotulados dirigidos (doravante mencionados apenas como grafos). Formalmente, um grafo G é uma tupla $G = \langle N_G, A_G, s_G, t_G, ln_G, la_G \rangle$ onde N_G é um conjunto de *nodos* (ou *vértices*), A_G é um conjunto de *arcos* (ou *arestas*), s_G e t_G: $A_G \to N_G$ são funções de atribuição de *origem* e *destino* para arestas, e $ln_G : N_G \to LN_G$ e la_G: $A_G \to LA_G$ são funções que associam *rótulos*, respectivamente, para nodos e arcos.

O conceito de subgrafo refere-se a um grafo ser parte de outro, de modo que vértices e arestas de um grafo estejam contidos em um segundo grafo. Em termos formais, supondo que G e H são grafos, H é subgrafo de G se $N_H \subseteq N_G$, $A_H \subseteq A_G$, $s_H = s_G|A_H$, $t_H = t|A_H$, $ln_H = ln_G|N_H$ e $la_H = la_G|A_H$. A partir dessa definição é possível definir um subgrafo específico e trabalhar apenas com uma parte de um grafo maior.

Um morfismo entre grafos (desconsiderando o caso do homomorfismo) é uma bijeção entre vértices com preservação de arestas. Um morfismo total $m : G \to H$ do grafo G para o grafo H é um par de mapeamentos totais $m = \langle m_N : N_G \to N_H, m_A : A_G \to A_H \rangle$ preservando origens, destinos e rótulos, ou seja, mapeamentos satisfazendo $m_N \circ t_G = t_H \circ m_A$, $m_N \circ s_G = s_H \circ m_A$, $ln_H \circ m_N = ln_G$ and $la_H \circ m_A = la_G$. Um morfismo de grafos é um *isomorfismo* se m_N e m_A são mapeamentos bijetivos. Grafos dirigidos rotulados e morfismo total de grafos podem ser combinados para formar a categoria **Grafo**, na qual grafos e morfismos são, respectivamente, objetos e setas.

2.5 Transformações de Grafo

A teoria de grafos realiza estudos sobre propriedades de grafos sem propor alterações em seus vértices e arestas. Em contrapartida, as Transformações de Grafo são operações que realizam mudanças estruturais por meio de aplicação de regras ou produções de grafo (EHRIG et al., 2015b). Uma produção de grafo (ou apenas uma produção) $p = (L, R)$ é composta por um grafo a esquerda L, um grafo a direita R e um mecanismo que sistematicamente especifique como L se transforma em R.

Outros termos encontrados na literatura para designar transformações de grafo são: sistemas de rescrita de grafo e gramática de grafos. Usualmente esses termos são utilizados como sinônimos de maneira intercambiável, a escolha de palavras normalmente se deve ao enfoque que se quer dar ao sistema, execução de algumas transformações específicas ou listar todas as possíveis estruturas geradas a partir de um grafo inicial.

O detalhamento da regra que transforma um grafo em outro e especificidades de sua aplicação caracterizam diversas abordagens, dentre as quais podem ser destacadas *Node Label Replacement Approach*, *Hyperedge Replacement Approach*, *Algebraic Approach*, *Logical Approach*, *Theory of 2-Structures* e *Programmed Graph Replacement Approach* (EHRIG et al., 2006).

A abordagem escolhida para essa dissertação é a *Algebraic Approach*, tratada daqui para frente como Abordagem Algébrica que surgiu no começo da década de 70 buscando generalizar as gramáticas de Chomsky de strings para grafos. Nessa abordagem, os grafos, conforme definidos anteriormente, são álgebras cujos conjuntos básicos são vértices e arestas sendo que a rotulação constitui as operações. Esta forma de trabalhar com sistemas de rescrita de grafos foi proposta por (EHRIG; PFENDER; SCHNEIDER, 1973) e baseada na teoria das categorias, trabalhando com homomorfismos e construções do tipo *pushout* na categoria **Grafo**.

Essa abordagem pode ser caracterizada pelo embasamento categórico para definições simples de três conceitos básicos: Regras de Transformação/Produções, matches/casamentos/morfismos (a ocorrência do lado esquerdo da produção em um grafo específico) e aplicação de regras/derivações diretas. A principal vantagem dessa opção é justamente seu embasamento na teoria das categorias, que permite desenvolver provas, que normalmente não dependem da estrutura dos objetos, com esforço menor em relação a outras abordagens (ROZENBERG, 1997).

Consideremos uma produção genérica $p : L \rightarrow R$, chamamos L e R de grafo esquerdo (*left-handside* ou LHS) e direito (*right-handside* ou RHS) respectivamente, essa produção é uma representação finita de derivações que caracterizam um conjunto potencialmente infinito. Suponhamos que um casamento m ocorreu no grafo G temos então que $G \overset{p,m}{\Rightarrow} H$ é uma derivação direta da aplicação de p em G que resulta em um grafo H. Uma forma intuitiva de ilustrar essa produção é que o grafo H foi gerado a partir da substituição de L por R em G.

Nessa etapa da definição das transformações é importante entender três questões: o que é um grafo, como se dá o processo de casamento e se é possível definir a substituição de L por R em G (ROZENBERG, 1997) A definição da categoria **Grafo** foi feita na seção anterior, então

nos resta discorrer sobre os dois outros pontos. Um casamento $m : L \rightarrow G$ dentro do contexto de uma produção de grafos é um homomorfismo mapeando vértices e arestas de L em G de modo que possíveis rotulações sejam preservadas. A substituição compreende o processo de deleção de todos os elementos (vértices e arestas) que constam em L porém não aparecem em R, bem como a criação de todos os elementos que não aparecem em L e aparecem em R.

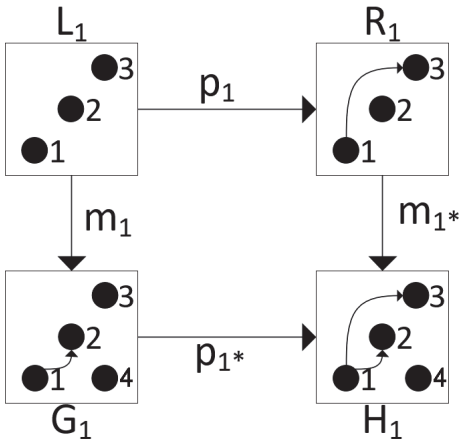

Figura 5 – Transformação de Grafos

Para ilustrar melhor os mecanismos que compõe uma transformação de grafos, vamos sair do âmbito abstrato e partir para uma instância concreta. Consideremos uma produção concreta $p_1 : L_1 \rightarrow R_1$ ilustrada na Figura 5, verifique que o grafo L_1 é formado pelo conjunto de vértices $v = (1, 2, 3)$ e pelo conjunto vazio de arestas $a = \phi$, enquanto o grafo R_1 possui todos os vértices de L_1 porém possui também uma aresta ligando dois vértices. Dado o grafo G_1, o processo de casamento $m_1 : L_1 \rightarrow R_1$ identifica um homomorfismo que mapeia os nos 1, 2 e 3 do grafo L_1 aos nós de mesmo número no grafo G_1 que possibilita a aplicação da produção p_1. A aresta que consta em R_1 e não aparece em L_1 é então criada, gerando o grafo H_1, concluindo assim a transformação $G_1 \overset{p_1,m_1}{\Rightarrow} H_1$.

Conforme visto no exemplo anterior, o arco ligando os nodos rotulados por 1 e 2 (ver Figura 5) que consta no grafo G_1 porém não consta no grafo L_1 é mantido na aplicação da produção, porém isso não ocorre para todos os nodos não explicitamente definidos nos grafos a esquerda e a direita. Em uma situação de deleção de vértices, todas as arestas ligadas a ele são excluídas junto do procedimento.

Dados os conceitos já definidos, temos que uma gramática de grafos ϱ é formada por um conjunto de produções P e um grafo inicial G_0. Uma derivação dessa gramática é constituída por uma sequência de derivações diretas $p = (G_0 \overset{p_1}{\Rightarrow} G_1 \overset{p_2}{\Rightarrow} ... \overset{p_n}{\Rightarrow} G_n)$. Por fim, a gramática ϱ gera a linguagem $L(\varrho)$ que é constituída a partir de todos os grafos G_n gerados a partir de derivações da gramática.

Quando falamos em transformações de grafo dentro do contexto de sistemas computacionais, é comum utilizarmos um grafo para definir o estado atual do sistema e o conjunto de produções como suas transições, outros problemas comuns são a transformações de grafo con-

siderando situações com concorrência, sincronização e distribuição (ROZENBERG, 1997).

Derivações diretas podem ser executadas de forma sequencial, porém é possível trabalhar com composições de derivações de formas diferentes, que recebem o nome de derivações compostas. Derivações compostas podem utilizar técnicas de paralelização e amalgamação, porém elas estão fora do escopo dessa dissertação.

Com a formalização das categorias adesivas e a descoberta que **Grafo** se encaixa nessa categoria (LACK; SOBOCINSKI, 2004), novas formalizações e aplicações foram trabalhadas na abordagem algébrica. No livro de (EHRIG et al., 2015b) podemos encontrar aplicações em transformação de modelos para auxiliar no processo de desenvolvimento e no projeto de sistemas adaptativos, no qual as transformações podem representar os diferentes estados dos sistemas.

2.6 Sistemas Multiagentes

Embora a literatura não forneça uma definição universalmente aceita de agentes de software (RUSSELL et al., 2013; SHEHORY; STURM, 2014; WEISS, 2013), no escopo dessa dissertação vamos entender agentes como entidades abstratas que estão inseridas em um ambiente específico e que podem atuar sobre esse ambiente. Suas características principais são: Autonomia para buscar objetivos e atuar sobre o ambiente, Inteligência para se adaptar a situações novas e traçar planos de ação conforme sua base de crenças e sociabilidade para interagir com outros agentes de forma colaborativa ou competitiva (SHEHORY; STURM, 2014).

As implementações de agentes são variadas com base em sua fundamentação teórica. Dentre as possibilidades de implementação, destaca-se o BDI (*Belief Desire Intention*) (RAO; GEORGEFF, 1991) e possui como analogia o modelo do raciocínio prático humano (BRATMAN; ISRAEL; POLLACK, 1988). Dentro dessa ideia, o agente possui uma base de crenças (*Belief*) com base em sua capacidade de inspecionar o ambiente, desejos (*Desire*) são modelos ideais do ambiente aos quais o agente deseja atingir e as intenções (*Intention*) referem-se ao planejamento e as ações executadas pelo agente para modificar o ambiente e atingir seus objetivos.

Ao desenvolver sistemas computacionais cujas funcionalidades são desempenhadas por diferentes agentes, temos uma arquitetura de sistemas multiagente, de modo que os agentes trabalham de forma colaborativa ou competitiva para atingir objetivos específicos (SHEHORY; STURM, 2014). Nesse tipo de arquitetura, o número de agentes pode ser definido apenas em tempo de execução conforme a necessidade do sistema.

Sistemas multiagentes podem ser considerados soluções apropriadas para sistemas com características bastante particulares (WOOLDRIDGE, 2009), dentre elas podemos destacar:

- Ambiente dinâmico, não é possível antecipar as modificações realizadas no ambiente no qual o sistema está inserido, de modo que a autonomia dos agentes pode ser benéfica para o sistema;

- Descentralização de controle, em sistemas com funcionalidades bastante heterogêneas, pode ser difícil ou quase impossível projetar apenas um sistema que possua todas as características necessárias. Desse modo ao separar as responsabilidades entre diferentes agentes pode facilitar o desenvolvimento do sistema;

- Em situações nas quais precisamos modelar indivíduos independentes e autônomos, a analogia com agentes de software é direta e natural.

3 TRABALHOS RELACIONADOS

A pesquisa do estado da arte procurou identificar trabalhos relacionados à síntese de programas e transformação de software bem como as aplicações mais recentes de transformações de grafos. O modelo MITRAS é fundamentado na ideia que programas podem aplicar modificações em outros programas, ou nele mesmo. Esse conceito não é novo, inclusive existem especulações de que a ideia de programação automática foi conjecturada por desenvolvedores assim que as primeiras dificuldades apareceram (RICH; WATERS, 1993).

Mesmo considerando que não é objetivo do modelo MITRAS trabalhar com programação automática (área de pesquisa que tem por objetivo sintetizar algoritmos diretamente em código fonte a partir de uma descrição do problema), dada a semelhança de objetivo e relevância da área, é importante conhecer as abordagens utilizadas nesse tipo de pesquisa. Existem diversas publicações e projetos que trabalham com programação automática. Nesse contexto abordagens diferentes são utilizadas, podem ser destacadas algumas técnicas mais comuns, tais como: representação de conhecimento (RICH; WATERS, 1988), programação por exemplos (KATO; IGARASHI; GOTO, 2016), programação genética (XIAO et al., 2012; IGWE; PILLAY, 2013; REFORMAT; XINWEI; MILLER, 2003) e mineração de conceitos (TALANOV; KREKHOV; MAKHMUTOV, 2010). Apesar desse esforço de pesquisa, os resultados concentram-se no auxílio a desenvolvedores (RICH; WATERS, 1988) ou em aplicações acadêmicas (WAKATANI; MAEDA, 2015), de modo a ficarem muito restritas a experimentos de laboratório. Existe uma expectativa que essas funcionalidades estejam disponíveis em ferramentas comercias conforme os resultados dessa linha de pesquisa fiquem mais maduros (MCLAUGHLIN, 2006), embora relatos mais atuais ainda mostrem falhas nesse tipo de tecnologia (XIN; REISS, 2017).

A área de transformação de programas difere da programação automática, pois o ponto de partida é um sistema em funcionamento e não um enunciado de uma necessidade, característica comum ao modelo proposto nessa dissertação. A partir de um software em funcionamento, ocorre um determinado processo de transformação para a derivação de novas funcionalidades. A pesquisa em transformação de software possui aplicações em alguns problemas como: melhoria na arquitetura de sistemas (TAHVILDARI; KONTOGIANNIS, 2002; AMIRAT et al., 2012), eliminação de trabalho repetitivo (DE ROOVER; INOUE, 2014; ROLIM et al., 2017) e processos de reengenharia de sistemas (CORDY et al., 2001; TAHVILDARI; KONTOGIANNIS, 2002). Com enfoque para aplicações e ferramentas usadas por desenvolvedores.

Considerando a expressividade de grafos na representação de artefatos ligados ao desenvolvimento, projeto e diagramação de software, existem algumas ferramentas que suportam a modelagem, análise e a simulação de transformações de grafo em diversas abordagens, inclusive a abordagem algébrica. Algumas ferramentas representativas nesse âmbito são: AGG (TAENTZER, 2003) utilizado para transformações gerais de modelos e o ActiGra (MEHNER-HEINDL; MONGA; TAENTZER, 2013) com resultados consistentes em transformações aplicadas ao diagrama de atividades do UML. Apesar dessas ferramentas oferecerem uma opção sólida para

auxiliar em transformações de grafo, nessa dissertação foi necessário flexibilidade nas transformações, pois informações do grafo LHS e RHS tais como nome do campo e local onde ele se encontra no sistema são fornecidas durante a execução, apenas após obter essas informações que é possível realizar uma aplicação de uma transformação. Em decorrência disso optou-se por não utilizar ferramentas de apoio e codificar os mecanismos de transformação.

Existem também relatos de aplicações de transformação de grafos na modelagem de sistemas auto adaptativos (EHRIG et al., 2015a). Esse tipo de sistema tipicamente necessita se adaptar a mudanças no ambiente mantendo algumas propriedades básicas. No cenário apresentado pelo trabalho, foi possível modelar tanto os estados do sistema usando grafos quanto suas transições a partir de sinalizações e transformações de grafo. Um ponto importante é que nesse trabalho, todos os estados e regras de transição eram conhecidos de antemão de forma a possibilitar o projeto das transformações.

Ainda sobre transformações de grafo, a abordagem algébrica rendeu alguns resultados consistentes na integração de modelos baseada em *Triple Graph Grammars* (ANJORIN; LEBLEBICI; SCHÜRR, 2016). Essa abordagem permite a criação de regras bidirecionais que podem garantir que mudanças realizadas em um modelo sejam replicadas simultaneamente para um segundo modelo, promovendo a consistência entre eles (KINDLER; WAGNER, 2007).

A SPLE (*Software Product Line Engineering*) é uma prática de desenvolvimento de software que visa maximizar o reuso de artefatos, tentando minimizar o trabalho em desenvolver software que já foi escrito. As vantagens de reutilizar software são bastante conhecidas, pois são utilizados componentes já testados, validados e, possivelmente, já otimizados (RADOSEVIC; OREHOVACKI; MAGDALENIC, 2012). Embora a linha de produto de software trabalhe extensivamente com reuso, não é trivial encontrar uma abordagem prática para automatizar o reuso dos componentes de software, nem uma análise para verificar quais são as possibilidades de sintetizar novos sistemas a partir de uma biblioteca de componentes disponível em tempo de execução.

Outra abordagem para a engenharia de linha de produto de software é a abordagem “system family”, que trata a linha de produto como uma família de sistemas (WEISS; LAI, 1999), por exemplo: existe uma família de sistemas que são browsers, para criar um sistema desses existe uma abordagem para lidar com a engenharia do domínio (aspectos comuns a família, renderizar páginas html por exemplo) e engenharia de aplicação (aspectos específicos de um membro da família, capacidade de aceitar plugins por exemplo).

Um conceito interessante dentre as ferramentas que trabalham com alguma forma de síntese de programas é a introspecção de código (RADOSEVIC; OREHOVACKI; MAGDALENIC, 2012), esse conceito dita a forma como um software gerado/sintetizado automaticamente pode explanar as etapas e inferências utilizadas para gerar o resultado em questão, é a forma do programa justificar como chegou a um resultado específico.

É possível encontrar metodologias de geração de código fonte para linguagens de script que se baseiam na ideia de criar frames compostos por três objetos, Especificação, Configuração e

Template (RADOSEVIC; OREHOVACKI; MAGDALENIC, 2012). Os objetos indicados são definidos com linguagens de domínio específica (DSL), de modo que a definição da especificação fica formatada como um documento chave=valor, a configuração é definida em uma linguagem similar a notação XML e os *Templates* são definidos a partir de tags HTML. Tendo esses 3 objetos como entrada, são aplicadas transformações ao *template* fornecido, até que a especificação seja atendida, gerando na prática uma árvore de busca. Como resultado, eles são capazes de gerar uma aplicação WEB e introduzem o conceito de introspecção de código, no qual o sistema é capaz de apontar quais os elementos que levaram a criar o código que está sendo executado.

Considerando a abordagem de algoritmos genéticos, é possível encontrar alguns resultados práticos (IGWE; PILLAY, 2013). Sua abordagem é diferenciada pois ele opta por explorar as possibilidades de combinar comandos básicos de programação (if,for, atribuição e operadores aritméticos) com intuito de chegar a resolução do problema. Para as possibilidades de combinações desses comandos é dado o nome de espaço de programa e essa exploração é feita usando algoritmos evolutivos. Como resultado dessa pesquisa foram gerados programas de forma automatizada para resolver 10 problemas de programação tipicamente aplicados nas disciplinas iniciais de um curso de ciência da computação

Tabela 2 – Comparativo de Trabalhos Relacionados

	Formal	**Toy Problem**	**Programação Automática**	**Modelo**	**Código Fonte**	**Para usuários**
MITRAS	X			X	X	X
Kato, Igarashi, 2016		X	X		X	
Xiao et al 2012		X	X		X	
Wakatani, 2015		X	X		X	
Amirat, 2012		X		X		
Rolim et al, 2017		X	X		X	
Ehrig et al. 2015	X			X		
Anjorin, et al 2016	X	X		X		

Uma compilação dos trabalhos relacionados mais representativos, comparando com a proposta do modelo MITRAS, pode ser verificada na Tabela 2, nessa tabela procuramos destacar características como, escopo das transformações (no nível de modelos, no nível de código fonte ou em ambos), uso de programação automática, uso de formalismos, experimentação com *Toy Problem* e público alvo (usuários ou desenvolvedores). Esse conjunto de atributos ajuda a situar a presente dissertação frente os demais trabalhos.

O modelo MITRAS difere dos trabalhos relacionado em 4 aspectos principais: a) Não é uma ferramenta de programação automática, pois não é realizada síntese de algoritmos dentro de métodos. O MITRAS trabalha apenas com transformações de alto nível, mudança de classes, atributos e relacionamentos do sistema; b) Manipulação de software tanto em nível de modelo quanto em nível de código fonte, mantendo a sincronização entre ambos via extração de mo-

delo e injeção diferencial de código fonte; c) Uso de componentes do sistema que normalmente não são citados, como arquivos de configuração e interfaces web, que não estão diretamente associadas com código fonte; e d) Não se enquadra como uma ferramenta convencional de modelagem e transformação voltada a programadores e projetistas, os modelos de grafo extraídos são apenas abstrações do sistema. O MITRAS almeja ser um mantenedor do sistema, possibilitando que um usuário com conhecimento do domínio da aplicação mas sem conhecimento técnico em computação possa solicitar algumas modificações diretamente ao sistema.

Da mesma forma, é importante salientar que os trabalhos relacionados cumprem com a função de disponibilizar ferramentas de transformação para desenvolvedores e apresentam resultados interessantes no escopo de *toy problems*. Contudo, não foi identificado nos trabalhos relacionados uma tentativa de viabilizar que essas ferramentas de transformação sejam utilizadas por usuários ou experimentações em sistemas comerciais, duas questões que são diferenciais para essa dissertação e sob as quais os resultados são trabalhados.

Conforme pode ser verificado na Tabela 2, a maioria dos trabalhos relacionados pesquisados é validado considerando algum "Toy Problem", que são problemas simplificados criados apenas para servir como objeto de estudo de alguma nova solução. Em contrapartida, nessa dissertação a experimentação é realizada no repositórios de código *open source* do sistema OpenMRS, software orientado a objetos e em uso em diversos países (mais detalhes sobre o processo de escolha do sistema em 4.2.1), esse repositório foi processado para extração de um modelo do software, conforme será detalhado em 4.2.2.

4 MODELO MITRAS

Ao longo deste capítulo será apresentado o modelo MITRAS. Primeiramente será realizada uma categorização de tipos de sistemas transformacionais e como o modelo MITRAS se insere nessa classificação, depois será apresentado o modelo computacional contemplando todos os módulos do MITRAS e a base de código fonte *open source* escolhida para a implementação, a arquitetura do protótipo e por fim a apresentação da formalização do modelo de grafos e transformações.

4.1 Categorização de Sistemas Transformacionais

A proposta de classificação de ferramentas de transformação definida por (KAHANI et al., 2018), coleta informação referente a diversas ferramentas de transformação de modelo e realiza uma classificação conforme o sua aplicação, as classificações propostas pelo grupo são **M2M** (*Model to Model*) na qual um modelo é convertido em outro modelo, **M2T** (*Model to Text*) na qual um modelo é convertido para código fonte e **T2M** (*Text to Model*) na qual código fonte é convertido para modelo. Esse tipo de classificação é importante para diferenciar ferramentas de apoio a transformação de sistemas, porém para sistemas transformacionais que agregam mais de um tipo de transformação e cujas transformações não são o propósito final mas apenas um meio de chegar a um outro objetivo (como o caso do MITRAS), essa classificação não é aderente.

Caso tentássemos classificar o MITRAS conforme essa terminologia, teríamos que a etapa de extração de modelo (4.2.2) seria classificada como **T2M**, a etapa de transformação de grafos (4.2.5) como **M2M** e a etapa de injeção de código fonte (4.2.6) como **M2T**. Essa classificação diferenciada para cada uma das etapas se deve que a classificação de (KAHANI et al., 2018) usa como principal critério de classificação o tipo de artefato no qual as transformações são geradas.

Tendo em vista esses problemas com a proposta de (KAHANI et al., 2018), nesta dissertação definimos uma nova proposta de classificação para sistemas transformacionais aplicados a linguagens de programação orientadas a objetos. Nessa classificação, o principal atributo considerado para diferenciar sistemas transformacionais é a granularidade da representação do código fonte. Sistemas transformacionais que possuem uma capacidade de inspeção mais detalhada do código fonte recebem uma categoria de classificação de nível maior, em contrapartida sistemas com menor capacidade de inspeção de código fonte recebem uma categoria de nível menor.

Em uma primeira análise, parece plausível projetar sistemas de transformação que consigam representar todos os detalhes do código fonte. Em teoria, esse tipo de sistema poderia sintetizar qualquer função computável se tornando equivalente a uma Máquina de Turing em poder computacional[1]. O lado negativo de uma abordagem tão detalhada é que o sistema transformacional

[1]Gramáticas de grafo são modelos computacionais equivalentes a uma Máquina de Turing (MCBURNEY;

Tabela 3 – Categorias de Sistemas Transformacionais

#	Visibilidade	Capacidades
1	Classes	Criar novas classes
2	Métodos e Atributos	Criar novas classes, atributos e métodos baseados em padrões pré-definidos, integrar classes, atributos e métodos previamente existentes.
3	Dependências Estáticas	Deleção de classes, métodos e atributos que possuem dependências estáticas
4	Dependências Dinâmicas	Deleção de classes, métodos e atributos que possuem dependências dinâmicas
5	Algoritmo	Criar ou modificar código de qualquer método

possui a mesma complexidade da linguagem de programação que está sendo representada e, apesar de ser altamente expressivo, suas vantagens sobre a própria linguagem de programação seriam incertas. No outro extremo, existem sistemas de transformação que trabalham apenas com representações de alto nível do código, como o diagrama de classes de uma linguagem orientada a objetos. Nesse nível de abstração os detalhes de implementação são omitidos, por isso as possibilidades de transformação são mais limitadas.

A Tabela 3 mostra as categorias de sistemas transformacionais propostas. É importante salientar que as capacidades descritas nessa tabela são cumulativas, e.g. sistemas que se enquadram na categoria três também possuem funcionalidades das classes um e dois. No seu estágio atual de implementação, o MITRAS é classificado na categoria dois, possibilitando criação de novas entidades baseadas em padrões pré-definidos, mas não possuindo informações sobre dependências algorítmicas entre os métodos (ver introdução para tipos de transformação suportados). Contudo, como trabalho futuro o MITRAS será alterado para se enquadrar na categoria três, aperfeiçoando as técnicas de extração de modelo, com intuito de extrair também as dependências estáticas entre métodos.

Quanto maior for o nível de classificação de um sistema transformacional, mais expressivo e complexo o sistema se torna. Utilizando o modelo MITRAS e a base de código fonte que será detalhada no próximo capítulo, foram realizadas experimentações tanto nos níveis um e dois, sendo que os grafos resultantes em uma representação nível um possuíam 584 vértices e 684 arestas enquanto no nível dois uma representação do mesmo sistema demandou 5849 vértices e 8038 arestas. É importante enfatizar que, no nosso domínio de aplicação, alterar o MITRAS para comportar representações de métodos e atributos tornou o modelo de grafos uma ordem de magnitude maior considerando número de vértices e arestas.

SLEEP, 1989), porém aqui estamos nos referindo ao código gerado por consequência da transformação.

4.2 Modelo Computacional

Antes de realizar uma análise detalhada de cada um dos componentes, será apresentada uma visão geral do MITRAS. Nossa proposta é proporcionar um modelo para transformação de software que possa ser utilizado para permitir mudanças de sistemas em uso, sem a necessidade de programação explicita. A Figura 6 ilustra com um diagrama os processos que compreendem o modelo MITRAS, a linha tracejada indica que o passo não precisa ser executado em todas as transformações.

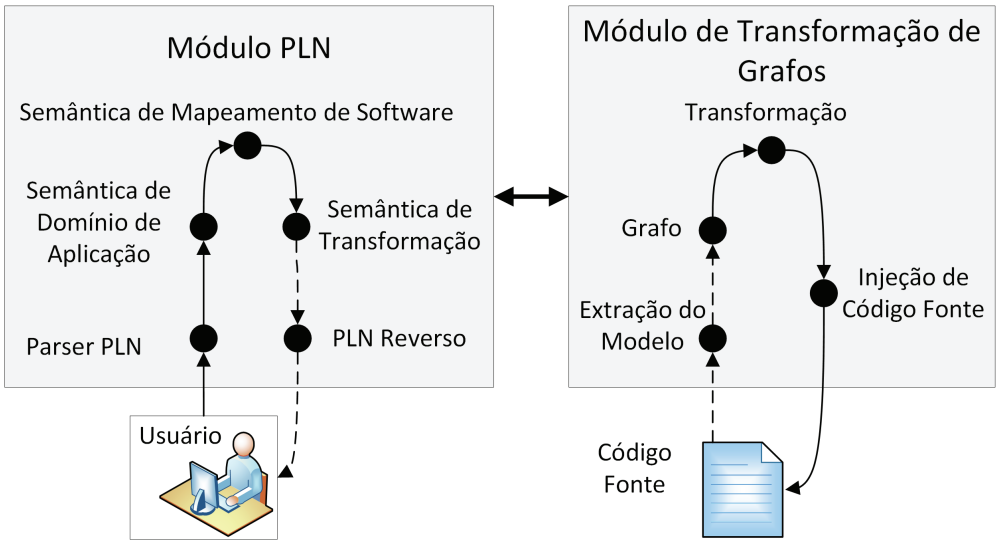

Figura 6 – Modelo MITRAS

Os arquivos fonte da aplicação, incluindo código fonte, arquivos de configuração e arquivos html, são o ponto de partida para a aplicação do modelo MITRAS.

Assim o primeiro passo é submeter o código fonte do sistema a um processo de de busca por entidades como classes, associações, dependências, arquivos de configuração e páginas html, essas informações são utilizadas na construção de um grafo tipado. Uma característica dessa etapa é assegurar que os vértices do grafo sejam construídos considerando informações de localização específicas, possibilitando a navegação do grafo para o código fonte. Dado que as transformações são aplicadas no nível do grafo, o procedimento de extração necessita ser aplicado uma única vez para a construção do grafo, mas caso seja necessário ele pode ser aplicado um número arbitrário de vezes sem efeitos colaterais. Um exemplo de situação que tornaria necessário um novo processo de extração é uma nova versão da aplicação que torne o modelo de grafo antigo incompatível com a versão antiga.

O grafo extraído do projeto é composto por informações encontradas tanto na árvore de parse quanto em elementos do software que não são compilados mas influenciam no funcionamento do sistema, como arquivos de configuração, propriedades e páginas html. Como o grafo criado é dirigido, todas as arestas possuem uma determinada direção, que por sua vez compõe o significado da relação que representa, por exemplo, em uma relação de herança a direção da

aresta indica qual é a super classe. Rótulos também possuem informações importantes, eles carregam os identificadores de classes, páginas jsp, métodos e atributos. No grafo extraído, tanto a rotulação quanto a direção das arestas carregam a semântica do modelo.

A etapa de transformação é o cerne do modelo MITRAS. Usando a técnica do single-pushout, subgrafos específicos são localizados no grafo original via morfismos e são transformados de modo que novos vértices e arestas são criados com a funcionalidade solicitada.

As mudanças realizadas por transformações no modelo de grafos são, por fim, replicadas ao código fonte da aplicação, em um processo que foi denominado injeção de código fonte. Essa etapa é realizada identificando os elementos que foram manipulados pela transformação, arestas e vértices adicionados por exemplo, sendo que as operações equivalentes são realizadas nos arquivos de código fonte e nos arquivos de configuração. Depois da injeção de código fonte, o sistema está pronto para os processos de build e deploy da nova versão.

4.2.1 Base de Código Fonte

Dadas as considerações formais e teóricas do modelo, a partir de agora teremos uma abordagem mais prática e funcional. Para analisarmos o modelo de maneira prática, possibilitando o uso de exemplos para elucidar as capacidades do modelo, necessitamos de um sistema real. Para tanto, procurou-se um software não trivial e arbitrário que possa servir com prova de conceito para o MITRAS, permitindo fazer uma boa avaliação das capacidades do modelo MITRAS. Os seguintes critérios foram adotados na procura deste sistema:

- Disponibilidade de código-fonte (projeto de código aberto)
- Uso de linguagem de programação orientada a objetos
- Existência de uma comunidade ativa de usuários e desenvolvedores
- Domínio de conhecimento diferente de Ciência da Computação

Esses critérios são importantes pois, projetos open source fornecem acesso ao código fonte, uma necessidade fundamental para aplicação do processo de extração que será definido na próxima seção com intuito de construir um grafo que realize uma representação abstrata do sistema, linguagens orientadas a objeto normalmente possuem uma arquitetura mais uniforme no sistema, ou seja, características arquiteturais tendem a se repetir em diversos locais no código fonte, (o que pode potencializar o uso de transformações mais genéricas) e, por fim, uma comunidade ativa significa que o software está ativo e continua relevante.

Escolher um sistema escrito em linguagem de programação orientada a objetos como Java em detrimento a linguagens cujo paradigma é mais flexível (como Python), traz vantagens ao uso desse modelo, visto que algumas transformações podem ser baseadas em design patterns que se beneficiam de uma arquitetura mais uniforme do sistema permitindo que transformações

mais gerais sejam definidas. Linguagens de programação de paradigma híbrido não impedem o uso do modelo, mas podem promover arquiteturas mais heterogêneas exigindo que mais transformações sejam criadas para atingir os mesmos objetivos.

Depois de pesquisar e avaliar uma ampla variedade de projetos de software com código fonte livre e aberto, foi escolhido o sistema para prontuário médico eletrônico OpenMRS (SEEBREGTS et al., 2009; MAMLIN et al., 2006). Ele é aderente a todos os critérios definidos e possui sua documentação disponível de forma online, ele é escrito em Java e possui em torno de 180.000 linhas de código, sendo utilizado em diversos países como África do Sul, Quênia, Haiti, Índia, Estados Unidos e China. O OpenMRS ainda também tem a vantagem de incluir um dicionário de sistema que define de forma consistente e completa a classificação de todos as entidades e propriedades de software usados no sistema. Além disso, a área médica, que é a área geral de aplicação do OpenMRS, tem o benefício adicional de ter muitas ontologias OWL de boa qualidade com definições claras para os termos e conceitos usados pelo OpenMRS.

4.2.2 Extração do Modelo

Repositórios de software open source agregam diversas informações diferentes, por isso é promissor aplicar técnicas de mineração para extrair dados que possam viabilizar novas abordagens na Engenharia de Software (HAN, 2017). No contexto do modelo MITRAS, técnicas de mineração de código fonte são aplicadas de modo a extrair um modelo de grafos do sistema a partir de regras sintáticas de código fonte e de arquivos de configuração.

Gramáticas formais são usadas para definir a estrutura sintática tanto do código fonte quanto de arquivos de configuração, de modo que esses tipos de arquivos sejam entendidos em termos de sentenças obedecendo a um conjunto de estruturas sintáticas e regras bem definidas (JACOBS; GRUNE, 1990). Certamente, códigos com erros de compilação não se enquadram em gramáticas e parsers de sua linguagem. No entanto, nessa dissertação assumimos que o software a priori é correto em termos sintáticos e semânticos, possibilitando que todos os arquivos fonte sejam submetidos ao processo de parse sem problemas.

Com intuito de ilustrar o processo de parse e a geração do modelo de grafo abstrato do sistema, será apresentado um exemplo de declaração de classe na linguagem Java e como as componentes do parse são transformadas em arestas e vértices do grafo. As figuras 7, 8 e 9 utilizam diagramas de sintaxe (também conhecidos como *Railroad Diagrams* ou diagramas ferroviários) para ilustrar a declaração de classes e suas dependências (BNF RULES OF JAVA, 2007). Esse tipo de representação exibe de forma gráfica uma gramática livre de contexto e, se comparada a notação de Backus-Naur, é igualmente expressiva (AHO; SETHI; ULLMAN, 2007).

Conforme ilustrado nos diagramas, o processo de extração do modelo a partir do código fonte é dirigido pela sintaxe da linguagem, de modo que estruturas encontradas no processo de parse viram vértices e arestas no modelo abstrato de grafos. As regras sintáticas da linguagem

compreendem aspectos suficientes para elaboração de sistemas transformacionais com visibilidade das dependências estáticas do sistema (ver seção 4.1), porém para o MITRAS a extração de declarações de Classes, Interfaces, Atributos e Métodos são suficientes.

Um diagrama Ferroviário pode ser entendido como um grafo, nesse caso o processo para gerar o grafo do MITRAS poderia ser pensado utilizando técnicas formais de transformação de modelos, nesse caso uma abordagem interessante seria o uso de triple graph grammars, mas está formalização está fora do escopo desse trabalho e é deixada para pesquisa futura. Nesse caso, cada elemento do diagrama é avaliado e produz uma aresta ou vértice equivalente. Por inspeção do grafo resultante na Figura 10, é possível identificar que os vértices foram criados com base em nós não terminais (identifier, identifier_name, class_name) dos diagramas de sintaxe apresentados nas figuras 7, 8 e 9. A combinação dos diagramas sintáticos das figuras 7, 8 e 9, com o grafo da Figura 10, constitui uma regra de extração de modelo dirigida por sintaxe. Em principio todo o processo de extração do modelo de grafos correspondente a um dado software é especifica por um conjunto destas regras. Isso permite que o grafo correspondente a classe declarada mantenha informações a respeito do nome da classe, métodos e atributos.

O processo de mineração necessita percorrer todos os arquivos, tanto código fonte quanto de configuração, para criar o modelo de grafos do software, durante esse processo também são buscadas informações que garantam a possibilidade de identificar a qual ponto específico dos arquivos essa informação representa (nome do arquivo e número da linha). Essas informações de rastreabilidade são agregadas ao grafo construído, possibilitando o trabalho da etapa de injeção de código.

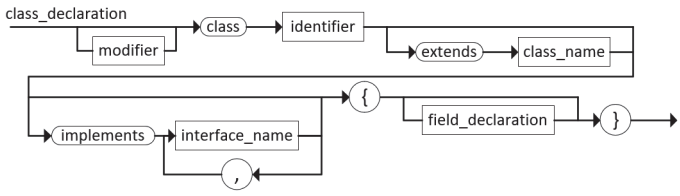

Figura 7 – Diagrama de sintaxe de declaração de classe

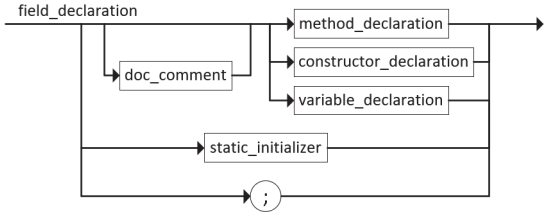

Figura 8 – Diagrama de sintaxe de declaração de atributo

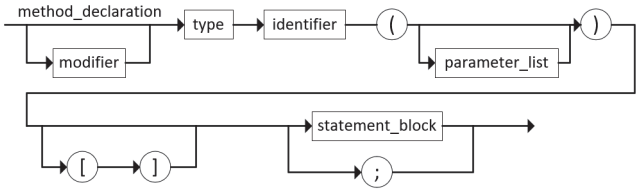

Figura 9 – Diagrama de sintaxe de declaração de método

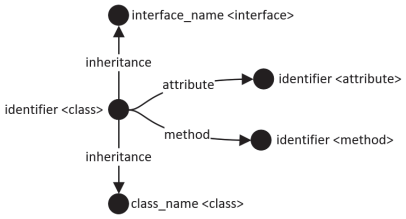

Figura 10 – Modelo extraído via regras de sintaxe

4.2.3 Avaliação de requisições via PLN

O processamento de linguagem natural no MITRAS incorporou o uso de três tecnologias distintas: o Stanford CoreNLP toolkit (MANNING et al., 2014), ontologias OWL e regras lógicas implementadas em Prolog. O toolkit de Stanford para PLN realiza parse de frases em inglês gerando diversas representações como POS tagging (TOUTANOVA; MANNING, 2000), name entity recognition(FINKEL; GRENAGER; MANNING, 2005) e geração de grafos de dependência(CHEN; MANNING, 2014), essas representações foram utilizadas pelo MITRAS.

Um grafo de dependências universais (CHEN; MANNING, 2014) é um grafo dirigido mostrando dependências gramaticais entre os termos de uma frase. Esse grafo foi considerado como um bom ponto de partida para realizar o processamento das requisições do usuário. Dessa forma, a partir de uma solicitação do usuário que foi processada pelo Parser de Stanford, obtemos o *grafo de dependências sintáticas*. Depois desse passo inicial, partes específicas da frase, representadas no grafo são consultadas na *Ontologia de Domínio de Aplicação* (ODA), esta ontologia possui informações referente a termos e e conceitos de domínio que ficam representados na interface de usuário do sistema. Por consequência disso, o grafo de dependências sintáticas é enriquecido com informação semântica formando o *grafo de dependência semântico*, que possui informações semânticas tanto do domínio quanto de interface de usuário. Após o processamento para identificação de quais termos da frase possuem mapeamento direto para a UI do sistema, o grafo passa a ser chamado *grafo de dependências mapeadas ao software*. Esse grafo final é então analisado para identificar qual é a intenção do usuário, ou, em outras palavras, o que o usuário quer fazer. Essa análise gera um *grafo de ação* que será utilizado no processo de identificação das transformações.

Para ilustrar o processamento de linguagem natural, será apresentado um exemplo de execução, salientado as etapas intermediárias e as estruturas de dados geradas durante o processo.

Suponha que a entrada do usuário seja a frase “Create a field called height in the name panel”, esta string é submetida ao parser de Stanford gerando o grafo de dependência sintática ilustrado na Figura 11.

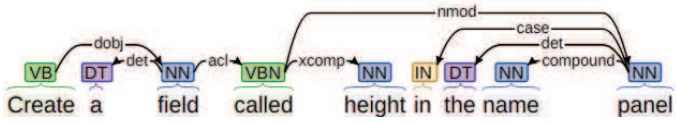

Figura 11 – POS tagger e Grafo de Dependências

Grafos de dependência podem ser expressados em Lógica de Primeira Ordem (LPO) por predicados monádicos como $vb(create)$, que define a categoria sintática (vb - verbo) da palavra “create”, predicados diádicos como $compound(panel, name)$, que definem uma relação de composição entre as palavras “panel” e “name”.

Quando o usuário interagir com o MITRAS, não esperamos que ele tenha conhecimento específico sobre programação, ou seja, ele não vai fazer solicitações baseadas em classes e métodos mas sim em elementos da interface de usuário. Dentro dessa ideia, para todo o X e Y que satisfizerem uma das duas expressões lógicas $(vb(X) \vee nn(X))$ and $(nn(X) \wedge nn(Y) \wedge compound(X, Y))$, a ontologia ODA é consultada para verificar se X ou a concatenação de X e Y fazem parte do sistema, se um ou mais termos forem encontrados na ontologia, então o grafo passa a ser enriquecido com informações de interface de usuário, resultando em um grafo de dependências semânticas, que também pode ser representado por predicados monádicos e diádicos da LPO.

A base da ontologia ODA é a *Clinical Measurement Ontology* (CMO) (SHIMOYAMA et al., 2012), que provê um vocabulário padronizado sobre tipos de medições clínicas usadas na avaliação de características fisiológicas e morfológicas. Conceitos e termos referentes a cuidados de saúde, setor administrativo e serviços compõem o dicionário de conceitos usados pelo OpenMRS. A *Ontologia de Dicionário de Conceitos*(ODC) é uma parte da ODA construída com base no dicionário de conceitos do OpenMRS que define os conceitos médicos usados em elementos como formulários, ordens, resumos, relatórios e conceitos que requerem persistência de dados e apresentação (UI). A parte final da ODA é a *Ontologia de Interface de Usuário*, formada por conceitos relacionados com elementos operacionais da interface de usuário, campos de tela, painéis e páginas e seus relacionamentos com as demais entidades do sistema ficam armazenados na ontologia, esses dados foram extraídos a partir da inspeção manual do sistema durante uma execução simulada. A Imagem 12 ilustra uma pequena parte da ontologia ODA, contendo conceitos relacionados a UI.

A ontologia ODA fornece informação sobre como identificar quais elementos da UI o usuário está mencionado e a qual conceito (entidade ou serviço) do OpenMRS esse elemento corresponde. É claro que existem diversas formas de se referir a um elemento de interface de usuário ou alguma entidade do sistema, dessa forma cabe a ontologia CMO fornecer sinônimos e formas análogas de descrever esses elementos. Após uma consulta a essa ontologia, nós sabemos

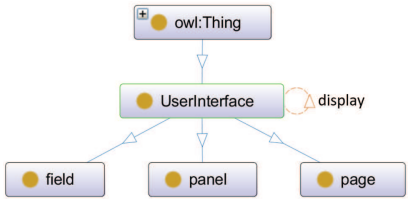

Figura 12 – Ontologia para mapeamento de interface de usuário

que “name panel” é um elemento de UI e “height” não está disponível na UI.

As informações que constam no grafo semântico são mapeadas as entidades como classes, atributos e métodos Java, considerando um identificador único comum ao modelo de grafos e as ontologias, esse identificador é uma propriedade anotada chamada *Id*. Esse grafo semântico devidamente anotado com os ids é chamado de Grafo de Dependências Mapeadas. A anotação destes *ids* estabelece uma correlção entre os nodos desse grafo e os elementos correspondentes no código fonte.

A última etapa no escopo do processamento de linguagem natural é inferir a intenção do usuário e construir o grafo de ação respectivo. Esse processo de inferência é executado a partir dos verbos presentes no grafo de dependências mapeadas, com o objetivo de identificar a intenção do usuário. Essa inferência utiliza a *Ontologia de Verbos para Ação* (OVA) que identifica quais verbos e quais estruturas de frase podem ser consideradas solicitações de transformação. Além disso, a ontologia OVA define a forma canônica ou padronizada de um verbo que será utilizada para especificar qual ação será feita. Isso é importante pois para acionar o módulo de transformações, apenas formas canônicas são aceitas, o grafo de ação é construído utilizando regras lógicas que expressão condições suficientes e necessárias para inferir uma ação de transformação. Como por exemplo a regra lógica (4.1) exprime as condições para inferir que o usuário deseja criar um campo em um formulário específico.

$$\begin{aligned} vb(create) \wedge dobj(create, attr) \wedge acl(attr, named) \wedge \\ xcomp(named, F) \wedge nmod(call, P_1) \wedge compound(P_1, P_2) \wedge \\ concat(P_1, P_2, P) \wedge softid(P, O) \rightarrow \\ has(O, F) \wedge panel(O) \wedge field(F) \wedge in(O) \wedge notin(F) \end{aligned} \quad (4.1)$$

A constante *create* na regra lógica (4.1) é a forma canônica de qualquer verbo relacionado a criação de um novo campo (outras possibilidades seriam “insert”, “add”, etc.). Da mesma forma, *attr* é o identificador canônico de um novo atributo, campo ou variável, enquanto *named* é o identificador padrão para um adjetivo (acl - adjectival clause) identificando o nome da entidade (outras possibilidades “called”, “labeled”, etc.).

As variáveis F, P_1 e P_2 são palavras da solicitação do usuário, P é a palavra formada pelo predicado de concatenação $concat$ e O é o id correspondente ao termo P, conforme obtido

da ontologia ODA pelo predicado $softid$. Os demais predicados compõem o grafo de dependência: vb é um predicado monádico de POS que indica que a palavra é um verbo, $dobj$, acl, $xcomp$, $nmod$ e $compound$ são predicados que definem as dependências universais. É importante salientar que a regra mostrada é apenas um exemplo utilizado no processo de inferência. O modelo MITRAS prevê capacidade para um número arbitrário de regras lógicas para manipular realizar inferência sobre as solicitações do usuário.

A parte direita da regra(4.1): $has(O, F) \wedge panel(O) \wedge field(F) \wedge in(O) \wedge notin(F)$ é o grafo de ação representado em predicados da Lógica de Primeira Ordem, que será encaminhado para o módulo de transformação de grafos. O *grafo de ação* contém informação que será usada para encontrar a transformação adequada que deve ser aplicada ao software. O predicado $in(O)$ indica que o vértice O necessita estar presente previamente no modelo de grafos do software, em contrapartida, $notin(F)$ indica que o vértice F não pode ocorrer no grafo atual.

É importante ressaltar que, a etapa de processamento de linguagem natural do MITRAS interage com o usuário até ser possível coletar informações necessárias para a identificação inequívoca de uma regra. A identificação de um conjunto de regras e sua aplicação concorrente estão fora do escopo dessa dissertação embora seja um interessante ponto para pesquisas futuras.

4.2.4 Modelo Abstrato de Grafos

O modelo abstrato de grafos do MITRAS deve ser capaz de representar os detalhes do código fonte estritamente necessários ao processo de transformação, além de manter a navegabilidade para o código fonte e permitir o uso de técnicas de reescrita de grafos. O modelo de grafos é detalhado nos próximos parágrafos.

O diagrama de classes de UML, que é capaz de representar todas as classes de um sistema orientado a objetos bem como suas relações é um bom ponto de partida para o modelo de grafos. Classes, atributos e métodos são identificados por técnicas de parse durante a etapa de mineração e geram vértices do grafo com rotulação adequada, as relações entre classes e as dependências entre métodos geram arestas com sua rotulação correspondente. Todas classes e relações encontradas no diagrama de classes, como uso, herança, associação e agregação são representadas por arestas no grafo, a Imagem 13 ilustra a representação de algumas classes e relações relações. Todas essas informações estão disponíveis de forma estática via *parsing*.

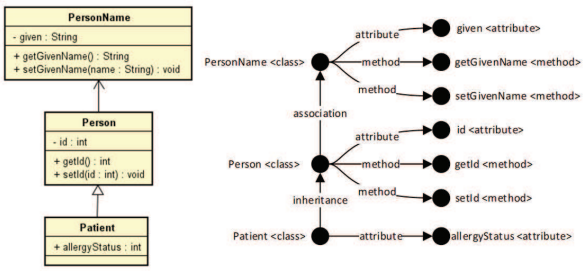

Figura 13 – Grafo do Diagrama de Classe

Apesar do diagrama de classes ser uma representação rica em informações sobre a arquitetura do sistema e possibilitar a busca pelos elementos equivalentes no código fonte, não existem informações suficientes nesse diagrama para dar conta de aspectos da interface de usuário e configurações do sistema que lidam com esses elementos. É necessário expandir o grafo para comportar também informações referentes a camada de apresentação do sistema e suas configurações.

O sistema OpenMRS, é uma aplicação web que conta com diversos módulos e interfaces de usuário. No que diz respeito ao modelo MITRAS, a camada de apresentação disponível por padrão no OpenMRS foi utilizada, essa implementação usa a tecnologia JSP. Dessa forma as páginas JSP são tratadas de forma semelhante a um arquivo HTML, sendo que suas dependências e elementos visual serão extraídos para compor o modelo de grafo. Páginas JSP, painéis, e campos de entrada das páginas são representados por vértices e suas relações são mapeadas como arestas com rotulação específica. A Figura 14 ilustra um exemplo de mapeamento de interface de usuário para grafo.

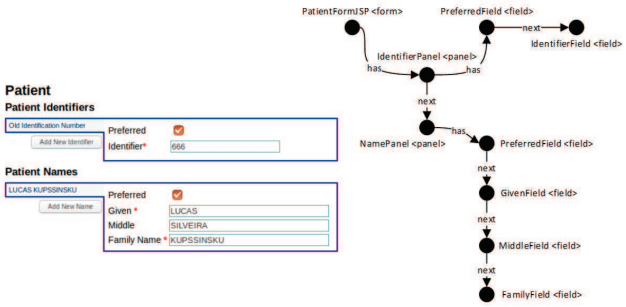

Figura 14 – Grafo do JSP

Os últimos artefatos que compõe o modelo de grafo são os arquivos de configuração do sistema. Existem diversos tipos de arquivos de configuração, a maioria deles aderente ao padrão XML, contudo o foco serão três tipos específicos de arquivos de configuração, são eles: Mapeamento Objeto-Relacional, Properties e Application Context. Estes arquivos, bem como seu equivalente no modelo de grafos, estão ilustrados na Figura 15.

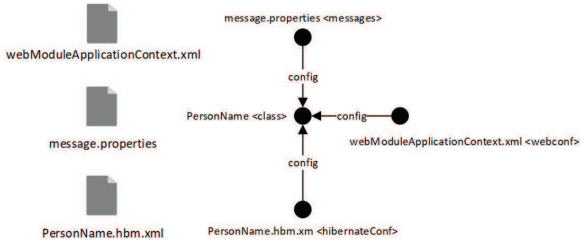

Figura 15 – Grafo do Arquivo de Configuração

O Mapeamento Objeto-Relacional dessa aplicação é realizado utilizando o framework Hibernate que, dentre outras formas, possibilita o uso de arquivos xml que descrevem como classes

Tabela 4 – Vocabulário Ontológico para Rotulação

Rótulo	**Tipo**	**Local**	**Significado**
form	UI	Vértice	Formulário
panel	UI	Vértice	Painel
field	UI	Vértice	Campo
show	UI	Aresta	Componente de UI mostra uma Classe ou Atributo
has	UI	Aresta	Componente de UI contém outro componente de UI
next	UI	Aresta	Próximo componente de UI em uma lista
class	OO	Vértice	Classe
interface	OO	Vértice	Interface
method	OO	Vértice	Método
attribute	OO	Vértice	Atributo
inheritance	OO	Aresta	Herança
use	OO	Aresta	Uma classe usa outra classe
association	OO	Aresta	Associação entre classes
config	Configuração	Aresta	Uma configuração modifica outro elemento
hibernateConf	Configuração	Vértice	Mapeamento objeto-relacional
messages	Configuração	Vértice	Mensagens
webConf	Configuração	Vértice	Configuração Spring
occlusion	Transformações	Aresta	Aresta para sinalização de oclusão

Java devem ser mapeadas para uma entidade persistente em um banco de dados relacional. Com intuito de operacionalizar o modelo, o interesse é identificar qual arquivo é responsável por mapear qual classe, portanto cada arquivo é portado ao grafo como um vértice e uma aresta liga os vértices que representam o arquivo e a classe Java.

Arquivos do tipo Property são repositórios de informação no formato chave-valor, eles podem ter diversos usos em sistemas. No OpenMRS, um dos usos de arquivos desse tipo é para associar rótulos a campos disponíveis na interface de usuário, uma espécie de dicionário que traduz nomes de classes e atributos para nomes mais amigáveis para o usuário final. Existe um arquivo desse tipo no OpenMRS e ele será representado com vértices e arestas identificados quais os atributos possuem uma rotulação específica.

O sistema estudado utiliza também o framework de desenvolvimento Spring, portanto teremos que considerar o arquivo Application Context no grafo pois ele pode indicar quais os atributos de um Java Bean podem ser visualizados na camada de apresentação e quais não podem. Este arquivo é representado com um vértice e é conectado aos Beans por arestas.

Dessa forma é definido um vocabulário ontológico que engloba as possibilidades de rotulação/tipagem para vértices e arestas do modelo abstrato de grafos. A Tabela 4 lista este vocabulário e a semântica específica de cada um dos tipos. É possível verificar que cada rotulação é aplicada ou em vértices ou em arestas do grafo, caracterizando as rotulações como elementos de interface de usuário, arquivo de configuração, transformação ou código orientado a objeto.

4.2.5 Transformações

Com o sistema representado em forma de grafo, conforme elucidado nas seções anteriores, o proximo processo do modelo consiste em definir como apresentar as transformações propostas. Quatro transformações são exemplificadas de forma a apresentar funcionalidades como: adicionar informações persistentes no banco de dados, oclusão de elementos de interface, mudança de ordem em campos da interface de usuário e criação de novas páginas a partir de design patterns, todas as transformações se sustentam na abordagem algébrica de *single-pushout* para reescrita de grafos.

Existem dois procedimentos importantes das transformações de grafo por *single-pushout* que permeiam todas os casos apresentados, o procedimento de *identificação do isomorfismo* e o procedimento de *aplicação de regras de transformação*. O procedimento de *identificação de isomorfismo* estabelece um isomorfismo entre os grafos do lado esquerdo (definido na transformação) e um subgrafo do modelo abstrato, esse isomorfismo identifica o local do grafo abstrato no qual a transformação deve ser aplicada. O procedimento de aplicação de regras de transformação, consiste da criação e deleção de arcos e nodos no subgrafo localizado pelo processo anterior, de modo que o grafo do lado direito da transformação assume o lugar do subgrafo encontrado. Basicamente as regras de transformação podem ser simplificadas pelos três casos a seguir: 1) Vértices e arestas que são identificados no grafo da esquerda e que não são identificados no grafo da direita são excluídos; 2) Vértices e arestas identificados em ambos os grafos são mantidos; 3) Vértices e arestas que são identificados no grafo da direita mas não são identificados no grafo da esquerda são criados.

No modelo MITRAS a abordagem adotada para a aplicação das transformações foge dos conceitos de gramáticas de grafo e adota uma abordagem mais funcional. Em gramáticas de grafo gerativas, poderíamos chegar a situação que um conjunto de regras é aplicado indefinidamente em um grafo, o que poderia trazer problemas de concorrência e de execuções que não terminam. Para contornar essa situação, as transformações são tratadas como funções, elas são identificadas e aplicadas ao grafo do modelo individualmente e sem possibilidade de repetição, nos casos de regras compostas (como na transformação quatro mostrada logo a seguir) as restrições de aplicação (ver seção 4.4) garantem que apenas uma regra seja selecionada.

No decorrer das próximas sessões, as transformações serão apresentadas e ilustradas com figuras, destacando em verde vértices e arestas criados.

4.2.5.1 Transformação 1 – Adicionar Campos

A primeira transformação diz respeito a funcionalidade de adicionar um novo campo persistente em um cadastro. Com intuito de realizar essa transformação, respeitando a arquitetura do OpenMRS, é necessário adicionar um atributo a uma classe específica e propagar essa mudança para os arquivos que tratam de interface de usuário e configurações. Do ponto de vista de uma

transformação de grafos, é necessário definir os grafos a esquerda e a direita da transformação.

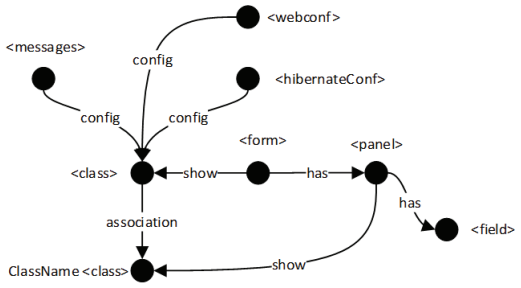

Figura 16 – Transformação 1 - Grafo da Esquerda

A Figura 16 ilustra o lado esquerdo da primeira transformação, é possível identificar que esse subgrafo comporta os relacionamentos entre arquivos de configuração e a relação da entidade com os elementos de interface de usuário. Ao aplicar esse grafo em um morfismo no modelo abstrato de grafo, teremos garantia de que foi identificado no sistema uma entidade com mapeamento objeto-relacional, caracterizando uma informação persistente no sistema.

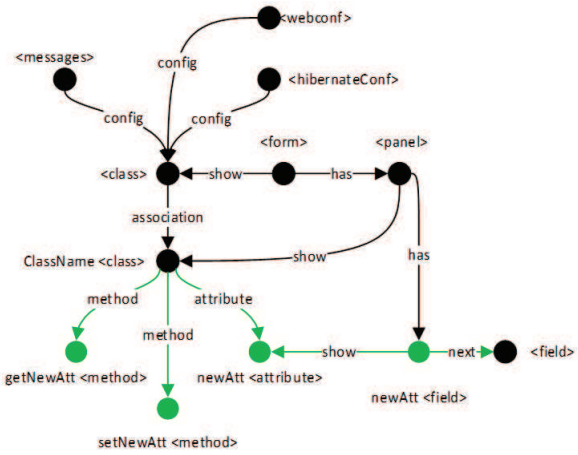

Figura 17 – Transformação 1 - Grafo da Direita

O lado direito da transformação é mostrado na Figura 17, com destaque para os arcos e nodos criados. Para a inclusão de uma nova informação persistente, é necessário que o atributo e os métodos de acesso sejam disponibilizados na classe e as amarrações com os arquivos de configuração sejam realizadas, os vértices "ClassName"e "NewAtt"representam respectivamente a classe buscada e o atributo que deve ser adicionado. Após a criação dos elementos necessários, o processo de injeção de código fonte é realizado.

4.2.5.2 Transformação 2 – Oclusão de Informação

O segundo caso de uso diz respeito a transformações para oclusão de campos na interface de usuário, de modo que informações possam ser ocultadas das páginas. No sistema em questão,

campos são renderizados em páginas JSP utilizando anotações no arquivo Application Context. Na prática, essa transformação precisa prover um mecanismo para desabilitar essa anotação no arquivo de configurações. Como o conteúdo do arquivo não fica exposto no modelo, é possível trabalhar com a adição de arcos entre o arquivo de configuração e o atributo da classe que se quer ocultar. Outra opção seria excluir o arco entre a classe e o elemento da interface gráfica, contudo a opção de adicionar um arco para representar a oclusão foi escolhida pois dessa forma nenhuma informação do grafo é perdida, facilitando o processo de desfazer a transformação.

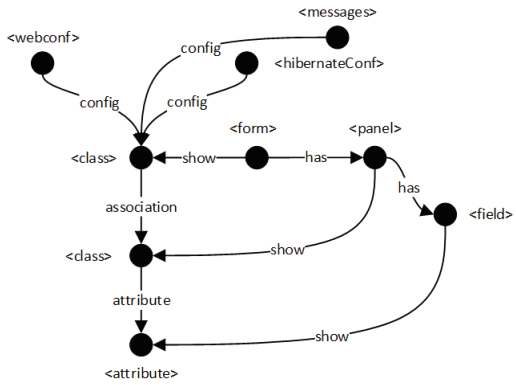

Figura 18 – Transformação 2 - Grafo da Esquerda

A Figura 18 mostra o lado esquerdo da transformação, evidenciando os elementos de interface e sua relação com as entidades. A Figura 19 ilustra o lado direito dessa transformação, exibindo o arco de oclusão adicionado. Caso seja necessário exibir novamente a informação que foi ocultada em outro momento, basta invertermos os lados esquerdo e direito da transformação para que o arco adicionado seja excluído.

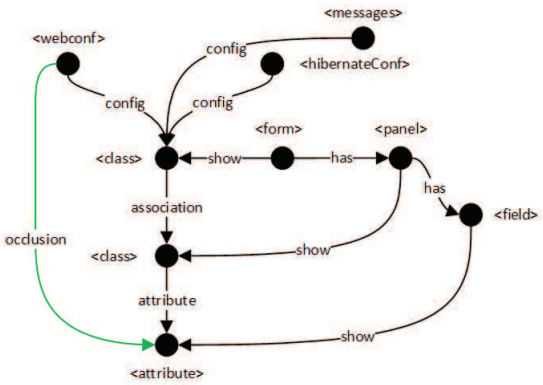

Figura 19 – Transformação 2 - Grafo da Direita

4.2.5.3 Transformação 3 – Reorganização de Interface de Usuário

A capacidade de customizar a interface de usuário, reordenando elementos conforme a necessidade, é abordada na terceira transformação. De modo que transformações de grafo são aplicadas para reordenar elementos de interface.

A interface gráfica é caracterizada por páginas html e jsp que constam no modelo abstrato de grafos, no caso de elementos que possuem uma ordenação na sua visualização, a ordem dos elementos (sejam eles campos ou painéis) ficam representadas de uma forma análoga a uma lista ligada, os vértices que representam os elementos da interface são ligados por arestas com rotulação específica e a direção dessas arestas dá a noção da ordenação dos elementos. A modificação da ordem de campos da interface é mapeada através de mudança de ordem de elementos em uma lista ligada, usando transformações de grafo.

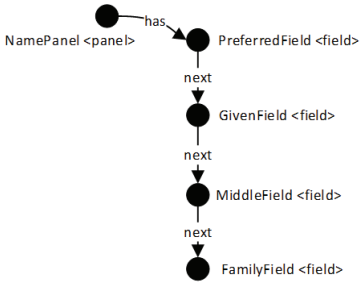

Figura 20 – Transformação 3 - Grafo da Esquerda

A representação da interface de usuário conforme uma lista ligada pode ser verificada na Figura 20. A Figura 21 por outro lado, exibe a reordenação já realizada, a mudança nas arestas de ligação indicam a mudança da ordem dos elementos na interface de usuário. Como resultado final o número de arestas e vértices do grafo não muda, visto que o mesmo número de arestas excluídas e criado novamente.

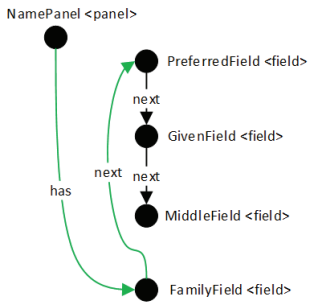

Figura 21 – Transformação 3 - Grafo da Direita

4.2.5.4 Transformação 4 – Implementação de Design Patterns

A quarta transformação é um pouco diferente dos demais. Enquanto os exemplos anteriores basicamente manipulavam entidades, criando alguns vértices e arestas conforme a necessidade, essa transformação realiza criação de diversos elementos no sistema, de modo que a parcela de informação criada no grafo é maior do que o próprio grafo original. Cabe lembrar que, essa transformação é tratada da mesma forma que as transformações previamente apresentadas.

Para atingir uma gama maior de transformações possíveis enquanto mantemos uma adequação a aspectos arquiteturais da aplicação, é possível incorporar uma biblioteca de padrões de projeto para ser manipulada pelas transformações. Neste exemplo, o padrão de projeto *abstract factory* será utilizado para fazer a criação de novas classes expondo apenas uma interface de comunicação comum.

Nessa transformação, o objetivo do usuário é criar uma nova página no sistema que resuma informações que constam em outros cadastros. Normalmente, seria necessário um desenvolvedor para codificar as novas classes e páginas para atingir esse objetivo, porém ao utilizarmos transformações de grafo é possível automatizar essa atividade.

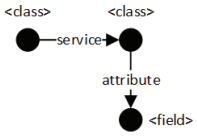

Figura 22 – Transformação 4 - Grafo da Esquerda

Como é possível identificar na Figura 22, o grafo do lado esquerdo dessa transformação é bastante simples, qualquer classe que está ligada a uma service class e que possui um campo qualquer pode acionar essa transformação. A aresta de serviço indica um relacionamento entre uma classe e um provedor de serviço para acesso a persistência de dados. Na arquitetura do OpenMRS, classes de serviço manipulam informações do banco de dados para entidades persistentes.

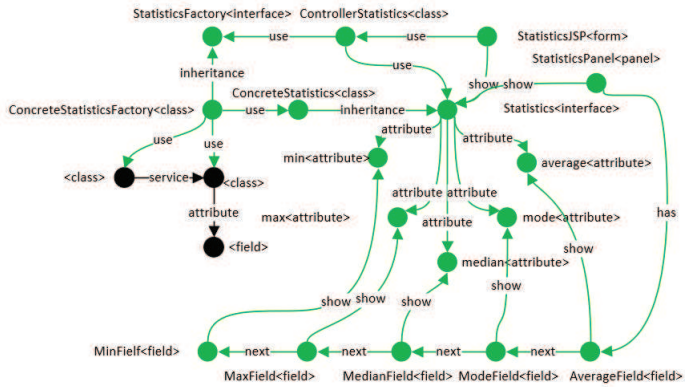

Figura 23 – Transformação 4 - Grafo da Direita

A etapa de derivação da quarta transformação cria classes análogas a proposta do design pattern Abstract Factory (GAMMA, 1995) e realiza a integração entre interface de usuário e acesso a banco de dados.

A transformação, conforme definida até aqui, atinge o objetivo de fornecer a nova funcionalidade desejada no sistema. Porém é possível ir um passo adiante, e projetar uma etapa anterior a essa transformação, essa etapa pode ser convenientemente projetada de modo a reutilizar o padrão de projeto implementado, minimizando a quantidade de código injetado e promovendo

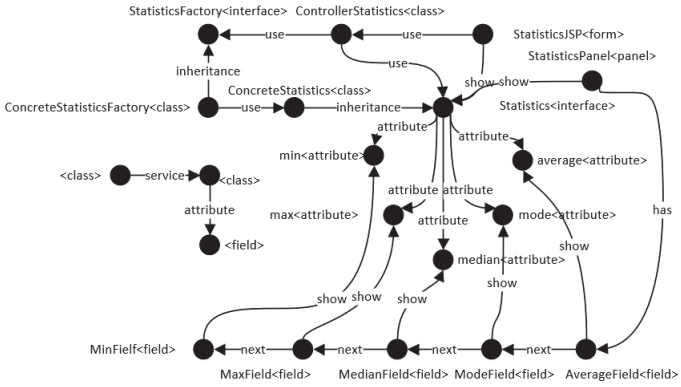

Figura 24 – Transformação 4.1 - Grafo da Esquerda

o reuso. O grafo do lado esquerdo dessa transformação, apresentado na Figura 24 identifica as classes do abstract factory e as classes concretas, a transformação então consiste na criação de dois novos arcos conectando as classes com seus respectivos factories, conforme ilustrado na Figura 25.

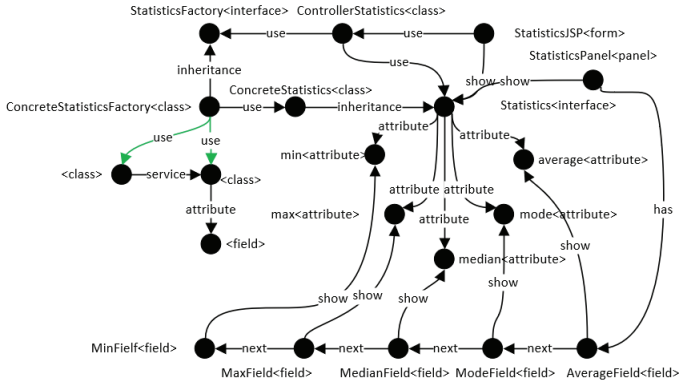

Figura 25 – Transformação 4.2 - Grafo da Direita

Dessa forma, é possível verificar que mesmo utilizando mecanismos automatizados de transformações, é possível trabalhar com reuso de código. Basta que os procedimentos de busca e de aplicação das transformações sejam organizados de forma a se aproveitar de código previamente escrito, conforme exemplificado na transformação quatro.

4.2.6 Injeção de Código Fonte

Para completarmos a análise dos componentes do modelo MITRAS, falta falarmos sobre o processo de injeção de código fonte. Toda a transformação de grafo é aplicada ao modelo de grafo abstrato, contudo essa é apenas uma representação intermediária do sistema criada para facilitar o uso de transformações com a abordagem algébrica, de fato o que se quer com uma transformação é que as mudanças sejam aplicadas no código fonte da aplicação, a esse processo foi dado o nome de injeção de código fonte.

Após uma transformação arbitrária ser executada, cada elemento novo do grafo, seja ele vértice ou aresta, deve ter um equivalente criado em nível de código fonte. Nos próximos parágrafos, as quatro transformações apresentadas anteriormente serão retomadas com enfoque no processo de injeção de código.

A primeira transformação apresentada, adiciona dois métodos e um campo em uma classe específica, isto é feito de forma trivial encontrando o arquivo fonte respectivo a classe modificada, a sintaxe para criação de métodos e campos é bem definida de acordo com a gramática da linguagem, então de posse de alguns templates de código fonte é possível automatizar esse processo, modificações necessárias aos arquivos de configuração podem ser inseridas no final deles.

A oclusão de campos é mapeada a partir da inserção de uma aresta de oclusão no grafo entre os vértices que representa, o arquivo de configuração e o atributo que deve ser ocultado. A injeção de código fonte é operacionalizada então pela inserção de um comentário na linha específica ao atributo dentro do arquivo de configuração, a operação inversa é realiza a partir da exclusão desse comentário.

Para mover um campo na interface de usuário, existe a necessidade de manipular diretamente o arquivo JSP dessa página, dado que a disposição dos elementos da interface está mapeada no grafo, a injeção de código precisa realizar um processo de recortar e colar para reposicionar os elementos na página JSP.

O fator principal da quarta transformação é o uso do padrão de projeto *Abstract Factory*. Esse padrão criacional é composto de quatro classes, duas do tipo *factory* e dois produtos, em ambos os casos uma das classes é abstrata e a outra é concreta (GAMMA, 1995). Todas essas classes necessitam ser criadas diretamente no código fonte a partir de *templates* pré-definidos.

4.3 Protótipo

Nas sessões anteriores a arquitetura do modelo MITRAS foi apresentada em um nível abstrato, foram abordados os conceitos empregados em cada uma das etapas do modelo, nesta seção será apresentada a arquitetura concreta do modelo após a implementação. Cabe lembrar que a arquitetura aqui apresentada é apenas uma das possíveis instanciações do modelo, diversas arquiteturas podem ser implementadas de acordo com as idiossincrasias do sistema com o qual se está trabalhando.

A Figura 26 fornece uma visão geral da arquitetura do modelo após implementação, nessa figura, o uso de retângulos com quatro ou dois cantos arredondados denota componentes desenvolvidos e integrados, respectivamente.

Na representação realizada, os limites do sistema foram representados por linhas tracejadas, inspecionando novamente a Imagem da arquitetura é possível perceber que as requisições feitas pelo usuário ao MITRAS passam sempre pela lógica de processamento PLN que processa as solicitações com auxílio da Ontologia de Vocabulários e do parser de Stanford, caso seja iden-

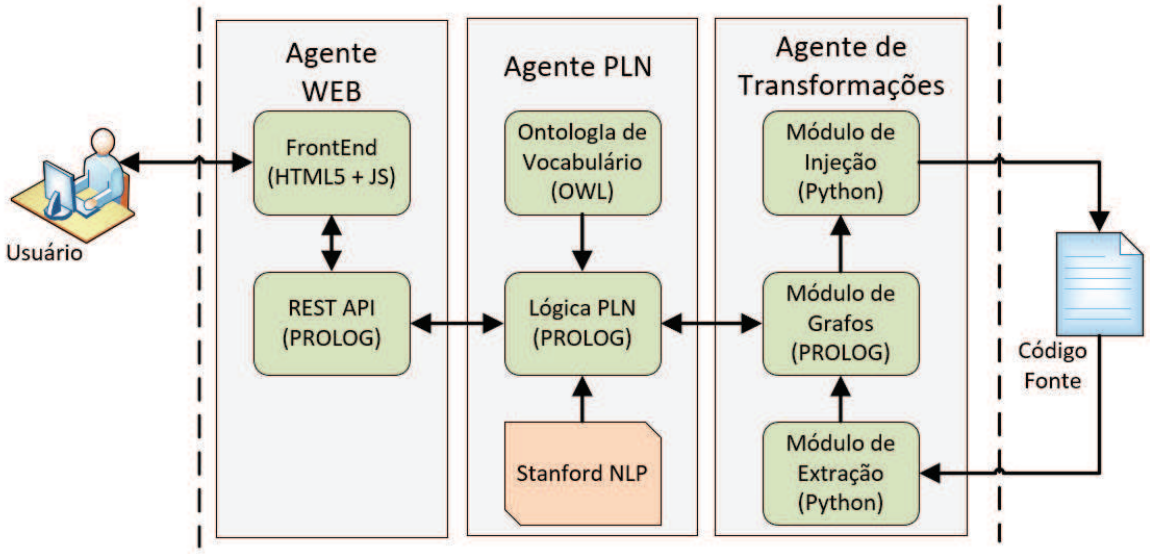

Figura 26 – Arquitetura Concreta do Modelo

tificada uma solicitação para realizar uma transformação no sistema, o módulo de Transformações de Grafo é acionado e por fim o módulo de injeção de software pode realizar alterações diretamente no código fonte da aplicação.

Conforme pode ser verificado, optou-se por trabalhar com diferentes tecnologias para abordar problemas diferentes. Nos módulos do sistema nos quais a prioridade era trabalhar de forma simbólica, manipulando estruturas de grafo ou estruturas escritas em linguagem natural foi utilizado PROLOG, uma linguagem de paradigma lógico que trabalha de forma bastante natural com esse tipo de problema. Para manipulação de código fonte e criação das árvores de parse para o código fonte da aplicação a linguagem python foi escolhida por fornecer uma interface mais amigável para leitura de arquivos de texto.

O módulo de extração é responsável por realizar a mineração do repositório e construir o modelo de grafos que vai alimentar o módulo de transformações. Na prática, os arquivos fonte, configurações e intefaces são lidos e geram um arquivo que serve como base de fatos para o programa PROLOG do módulo de transformações.

Um aspecto que não está destacado na Imagem mas que merece uma menção é a comunicação entre o parser de Stanford e a Lógica PLN em PROLOG. O parser de Stanford fornece uma interface de comunicação com a linguagem Java, portanto foi desenvolvida uma classe adapter em Java para consumir as apis de stanford e entregar os dados novamente ao PROLOG.

Ainda sobre a arquitetura, os retângulos em cinza denotam os agentes que compõe o sistema. Na fase atual de desenvolvimento, o MITRAS conta com três agentes responsáveis por tratar as requisições vindas do usuário através de uma interface web, realizar o parse e a avaliação da solicitação do usuário utilizando o parser de stanford e as ontologias desenvolvidas para esse fim e o agente de transformações que é responsável por criar as transformações nos níveis de modelo e código fonte.

Atualmente, o protótipo do MITRAS conta com uma base de regras de transformação capaz

Tabela 5 – Tempo de Execução

Transformação #	NLP	Transformação de Grafos	Injeção de Código	Tempo de Execução
1	35ms	146ms	331ms	512ms
2	27ms	160ms	215ms	402ms
3	33ms	103ms	224ms	360ms
4	38ms	217ms	355ms	610ms

de trabalhar com quatro tipos de transformação: 1) Adicionar campos: um campo persistente no modelo de dados é adicionado a uma entidade específica; 2) Oclusão de informação: um campo é ocultado do usuário; 3) Reorganização de Interface: A disposição de informações em uma tela é alterada; 4) Implementação de Design Pattern: Uma nova página é construída com base em elementos existentes do sistema combinados com um padrão de projeto. É importante salientar que o MITRAS não fica restrito a apenas essas quatro transformações, um número arbitrário de transformações pode ser definido dado que a nova transformação seja projetada e novas regras lógicas para identificação da intenção sejam definidas.

Os primeiros experimentos com o protótipo MITRAS foram conduzidos para avaliar funcionalidades do protótipo e tempo de execução de cada uma de suas transformações. A experimentação foi executada em laboratório usando um computador equipado com processador Intel i7 7500U e com 16Gb/2400MHz de RAM. A Tabela 5 resume os resultados dos experimentos.

A base de código fonte foi compilada com sucesso após a fase de Injeção de Código, contudo optou-se por não considerar o tempo de compilação nos resultados mostrados na Tabela 5, pois esse tempo permanece praticamente inalterado para todas as transformações. No hardware utilizado, o tempo de compilação do OpenMRS é de cerca de 6 minutos. Mesmo que optemos por adicionar esse tempo em nossa avaliação, o tempo mais longo de espera que um usuário necessita aguardar para a requisição de uma nova funcionalidade é em torno de 7 minutos.

O protótipo, em sua etapa de desenvolvimento atual, conta com uma base de transformações contendo quatro regras (conforme já foi apresentado anteriormente) sendo que cada regra conta com um conjunto de cerca de trinta regras lógicas para identificação da intenção do usuário a partir dos grafos gerados no processamento de linguagem natural. Essa quantidade de regras nos garantiu flexibilidade para identificar tanto as solicitações de transformação que são completas, quanto solicitações que exigem uma etapa de PLN reverso para questionar ao usuário sobre alguma informação que está faltando, e.g. o nome de um campo que deve ser adicionado a um cadastro.

O processo de mineração realiza leitura e *parse* dos arquivos de código fonte bem como de arquivos de configuração com intuito de construir o modelo de grafos do sistema. O processo de *parse* é análogo a um *parse* genérico de uma gramática livre de contexto, que possui complexidade de $O(n^3)$ no pior caso para uma string de n terminais, contudo, para os casos de linguagens de programação normalmente é possível trabalhar com parsers de tempo linear

(AHO; SETHI; ULLMAN, 2007). Dado que a árvore de parse do código é construída nessa etapa, o modelo abstrato de grafos pode ser construído de maneira direta, em tempo linear.

A etapa de transformação apoia-se nos formalismos de single-pushout para gramáticas de grafo. Conforme já foi detalhado em seções anteriores, para uma produção ser aplicada em um grafo, deve existir um morfismo entre os grafos do lado esquerdo e do grafo direito. Não se sabe se o problema de morfismos de grafos é resolvível em tempo polinomial, os algoritmos estado da arte para esse tipo de problema operam com complexidade $O(n * e!)$, fazendo com que essa etapa seja um ponto crítico do modelo MITRAS. Contudo, vale lembrar que o problema de morfismo de grafos instanciado na transformação envolve apenas um subgrafo definido no lado esquerdo da transformação, não todo o modelo de grafos.

Dado que informações referentes a localização de arquivos fonte ficam salvas em vértices e arestas do modelo, o processo de injeção de código fonte é aplicado por um mapeamento direto entre vértices e arestas contra arquivos. Como esse mapeamento é único, esse algoritmo roda em tempo linear $O(n)$ onde n é o número de vértices e arestas que serão adicionados ou modificados pela transformação.

4.4 Modelo Formal para Artefatos e Transformações de Software

Com intuito de trabalhar na formalização dos aspectos transformacionais do modelo MITRAS, consideraremos as definições da categoria **Grafo** que consta nas sessões 2.4 e 2.5. Retomando a formalização de grafos dirigidos rotulados temos que G é uma tupla $G = \langle N_G, A_G, s_G, t_G, ln_G, la_G \rangle$ onde N_G é um conjunto de *nodos* (ou *vértices*), A_G é um conjunto de *arcos* (ou *arestas*), s_G e t_G: $A_G \rightarrow N_G$ são funções que atribuem *origem* e *destino* aos arcos, e $ln_G : N_G \rightarrow LN_G$ e la_G: $A_G \rightarrow LA_G$ são funções de rotulação para nodos e arcos, respectivamente. *Vocabulários Ontológicos* finitos LN_G e LA_G fornecem, respectivamente, rótulos para nodos e arcos do grafo G. Esses vocabulários são extraídos de um conjunto de identificadores presente em uma ontologia formal $\mathcal{O}$. LN_G é formado por identificadores de conceitos (ou classes) e indivíduos presentes na ontologia $\mathcal{O}$, enquanto LN_A é formado por identificadores de papeis (ou propriedades) presentes em $\mathcal{O}$.

Um *morfismo total de grafos* $m : G \rightarrow H$ de um grafo G para um grafo H é um par de mapeamentos totais $m = \langle m_N : N_G \rightarrow N_H, m_A : A_G \rightarrow A_H \rangle$ com preservação de origens, destinos e rotulação, ou seja, mapeamentos que satisfazem $m_N \circ t_G = t_H \circ m_A$, $m_N \circ s_G = s_H \circ m_A$, $ln_H \circ m_N = ln_G$ e $la_H \circ m_A = la_G$. Um morfismo entre grafos é um *isomorfismo* se ambos m_N e m_A são mapeamentos bijetivos. Grafos rotulados dirigidos e morfismos totais de grafos podem ser combinados para formar a categoria **Grafos**, na qual grafos e morfismos são, respectivamente, objetos e setas.

Outro conceito importante na formalização é o *subgrafo*. A noção intuitiva de subgrafo é que um determinado grafo é parte de um grafo maior, de modo que seus vértices e arestas estejam contidos em um grafo original. Em termos formais, dados os grafos G e H, H é um

subgrafo de G, representado por $H \hookrightarrow G$, se $N_H \subseteq N_G$, $A_H \subseteq A_G$, $s_H = s_G|A_H$, $t_H = t|A_H$, $ln_H = ln_G|N_H$ and $la_H = la_G|A_H$. Um *morfismo parcial de grafo* $m : G \to H$ é um morfismo total $m' : dom(f) \to H$ de um subgrafo $dom(f) \hookrightarrow G$ para H. O subgrafo $dom(g)$ é chamado de domínio de g. Grafos e morfismos parciais constituem a categoria **Graph$^{\mathbf{P}}$**. A categoria **Graph$^{\mathbf{P}}$** possui pushouts para morfismos $r : L \to R$ e $g : L \to G$, assumindo que r preserva a rotulação (KOCH; MANCINI; PARISI-PRESICCE, 2001).

Propriedades estruturais de grafos podem ser alteradas por regras de produção (EHRIG et al., 2015b), que são mecanismos formalmente definidos que transformam um grafo original (ou mãe) em um grafo destino (ou filho). Diversas técnicas de transformação foram estudadas e podem ser utilizadas para realizar modificações em grafos(ROZENBERG, 1997), nessa dissertação será focada a abordagem algébrica, mais especificamente na abordagem de transformações por *Single Pushout* (SPO) (LöWE, 1993; CORRADINI et al., 1997; EHRIG; KORFF; LöWE, 1991). Uma regra de transformação em SPO é representada por uma *produção de grafo* $p : L \xrightarrow{r} R$, constituída pelo nome da produção p e por um morfismo de grafo r do grafo L para o grafo R. Os grafos L e R são chamados, respectivamente, de lados esquerdo e direito da produção p. A aplicação de uma produção $r : L \to R$ em um grafo mãe G é dada por um morfismo total $g : L \to G$. A partir disso, a derivação direta de G para um grafo filho H é produzida por uma construção do tipo pushout de r e g (ver Fig. 27)(EHRIG; KORFF; LöWE, 1991).

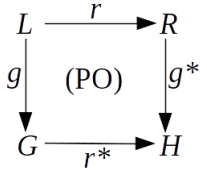

Figura 27 – Derivação Single-Pushout

Dado um morfismo parcial r e um morfismo total g, o grafo H, e os morfismos $r*$ e $g*$ podem sempre ser construídos seguindo os seguintes passos (EHRIG; KORFF; LöWE, 1991):

1. Construção de um grafo de "gluing" K. Este grafo é composto por todos os objetos (arcos e nodos) do $dom(r)$, que satisfazem as seguintes condições:

   (a) Para todo $x \in K$ e $y \in L$, se $r(x) = r(y)$ ou $g(x) = g(y)$ então $y \in K$.

   (b) Para todos os arcos $a \in K$, $s_L(a), t_L(a) \in K$

2. Construção dos grafos G_{r*} e R_{g*}. O grafo G_{r*} é construído a partir da deleção em G de todos objetos que são imagens de itens non-gluing w.r.t. g e todos os arcos, cujos vértices de origem ou destino tenham sido excluídos. Este tratamento é aplicado de forma análoga em R resultando no grafo R_{g*}.

3. Construção do grafo filho H (see Fig. 28). O grafo H é construído com os pushouts dos grafos G_{r*} e R_{g*} w.r.t. $r|_K : K \to R_{g*}$ e os morfismos $g|_K : K \to G_{r*}$, que são restrições dos morfismos r e g no grafo K. Os grafos G_{r*} e R_{g*}, e os morfismos $r|_K$ e $g|_K$ satisfazem a condição de gluing e o morfismo $r|_K$ preserva a rotulação, portanto o grafo H sempre pode ser construído utilizando construções de pushouts.

4. Construção dos morfismos $r*$ e $g*$ (ver Fig. 28). Ambos morfismos $r*$ e $g*$ coincidem com os morfismos produzidos pelos pushouts do grafo H. Portanto, o domínio de $r*$ é G_{r*} e o domínio de $g*$ é R_{g*}.

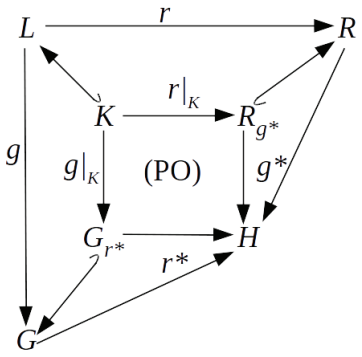

Figura 28 – Construção da Derivação Single-Pushout

É possível estender a noção de grafos rotulados com a introdução de *tipos*. Um *grafo tipado* define um conjunto de tipos, que podem ser usados para atribuição de tipos a vértices e arestas de um grafo. Comumente, essa tipagem é definida por um alfabeto de vértices e arestas que então definem os tipos dos objetos do grafo (EHRIG et al., 2006). A tipagem é realizada por um morfismo total do *grafo tipado* (ou *instancia* do grafo) no grafo de tipos, com preservação de tipagem. Esse morfismo total é chamado de *morfismo de tipagem*.

LN_G e LN_A, os conjuntos de rótulos para vértices e arestas de um grafo G, são vocabulários ontológicos finitos extraídos de identificadores de classes e indivíduos de uma ontologia formal particular $\mathcal{O}$. Esse trabalho assume que ontologias formais são definidas em *Lógicas Descritivas* (DL) (BAADER; HORROCKS; SATTLER, 2009, 2008), assim sendo, uma ontologia particular $\mathcal{O}$ é uma estrutura $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, na qual $\mathcal{T}$ é um conjunto finito de axiomas de terminologia (o *TBox* da ontologia) e $\mathcal{A}$ é um conjunto finito de axiomas asseridos (o *ABox* da ontologia) (BAADER; HORROCKS; SATTLER, 2008). Axiomas oriundos de $\mathcal{T}$ e $\mathcal{A}$ usam identificadores para conceitos/classes, responsabilidades/propriedades e indivíduos, respectivamente, dos conjuntos contáveis N_C, N_R and N_I (BAADER; HORROCKS; SATTLER, 2008). Note que, apesar de N_C, N_R e N_I serem conjuntos contáveis potencialmente infinitos, as restrição desses conjuntos com os identificadores presentes nos axiomas de $\mathcal{O}$, representados respectivamente por $N_C|_{\mathcal{O}}$, $N_R|_{\mathcal{O}}$ e $N_I|_{\mathcal{O}}$, são conjuntos finitos.

Isto permite que LN_G seja dividido em $LN_G^C = N_C|_{\mathcal{O}}$, o vocabulário de identificadores de conceitos/classes presentes em $\mathcal{O}$ e $LN_G^I = N_I|_{\mathcal{O}}$, o vocabulário de identificadores de indiví-

duos explicitamente asseridos na ontologia. Grafos tipados, no contexto desse trabalho, são grafos que contem vértices rotulados por identificadores de classes retirados do vocabulário LN_G^C. Grafos que contém vértices exclusivamente rotulados como identificadores individuais de LN_G^I são grafos de *instancia*. O conjunto de rótulos para arestas LA_G também é extraído do vocabulário de identificadores de responsabilidades/propriedades presente em $\mathcal{O}$, i.e. $LA_G = N_R|_{\mathcal{O}}$, mas esse conjunto não é utilizado para atribuição de tipos.

Dado um grafo de tipos TG, um par $\langle G, t\rangle$, onde G é um grafo e $t : G \to TG$ é um morfismo de tipagem, é chamado de *grafo tipado* TG. Preservação de tipos é uma extensão da preservação de rótulos definida pelo relacionamento entre instancias na ontologia $\mathcal{O}$:

(a) Para todos os vértices $x \in G$, cujos rótulos são identificadores de classe: $ln_{TG}(t_N(x)) \in LN_G^C$, precisa ser assegurado que $ln_G(x)$ é uma instancia de $ln_{TG}(t_N(x))$ in $\mathcal{O}$, i.e. $\mathcal{A} \models_{\mathcal{T}} ln_G(x) : ln_{TG}(t_N(x))$ (BAADER; HORROCKS; SATTLER, 2008; BLACKBURN; BENTHEM; WOLTER, 2006).

(b) Caso contrário, para todos outros vértices $x \in G$, tal que $ln_{TG}(t_N(x)) \notin LN_G^C$, a preservação de rótulo deve se manter: $ln_G(x) = ln_{TG}(t_N(x))$.

Para representar a transformação de artefatos de software será necessário suportar algumas condições adicionais na aplicação de regras de produção. Existem diversos tipos de *regras de aplicação* que podem ser utilizadas por regras de transformação de grafos (EHRIG et al., 2006), mas as regras de transformação utilizadas nesse trabalho usam as duas regras definidas a seguir:

- Condição de Aplicação Negativa (NAC): essa condição testa se algum grafo não está presente no grafo onde a regra será aplicada. NAC é definido pela regra $p : L \to R$ por um grafo N, tal que L é um subgrafo de $L \hookrightarrow N$. A condição é satisfeita por um grafo G, o grafo onde a regra será aplicada, se para algum subgrafo $N' \hookrightarrow N$, tal que L também é um subgrafo $L \hookrightarrow N'$, não existe um morfismo total de N' para G.

- Regra de Aplicação Distinta Individual (DIAC): esta condição verifica se os identificadores dos indivíduos atribuídos como rótulos de um grafo G onde a regra $p : L \to R$ deve ser aplicada são diferentes um do outro. DIAC é definido por uma regra $p : L \to R$ por *índices distintos* atribuídos a um subconjunto de vértices do grafo L por uma função parcial $di : N_L \to \mathbb{N}$. Esta condição é satisfeita por um grafo G e um morfismo total g de L para G se para todos os vértices x,y de L, tal que $di(x)$ e $di(y)$ é definida e $di(x) \neq di(y)$, uma das seguintes se mantém:

  (a) Se $ln_L(x) \neq ln_L(y))$ e $ln_L(x), ln_L(y) \in LN_G^C$, então seus rótulos no grafo alvo G devem ser diferentes: $ln_G(g(x)) \neq ln_G(g(y))$.

  (b) Se $ln_L(x) = ln_L(y))$ e $Ln_L(x) \in LN_G^C$, então seus rótulos no grafo alvo G devem coincidir: $ln_G(g(x)) = ln_G(g(y))$.

5 DISCUSSÃO SOBRE LIMITAÇÕES DO MITRAS

Nos capítulos anteriores foram apresentadas as capacidades do modelo para representação de sistemas e realização de transformações para sintetizar novas funcionalidades. O capítulo atual realiza a abordagem inversa, trabalhando com as limitações do projeto apresentado nessa dissertação.

Podemos separar as limitações do MITRAS em duas classes, limitações de modelo e de protótipo. As limitações de modelo são aquelas geradas a partir de escolhas feitas antes do desenvolvimento do protótipo, elas estão amarradas a decisões do projeto e a arquitetura do sistema. As limitações do protótipo, por outro lado, são inerentes a ideia de um protótipo como sendo uma aplicação para prova de conceito, ocorrem devido a simplificações feitas durante o desenvolvimento e que poderiam ser contornadas apenas com refinamento e mais tempo de desenvolvimento nesse protótipo. Serão aprofundadas as limitações intrínsecas ao modelo enquanto as limitações do protótipo são citadas de forma mais sucinta.

5.1 Limitações do Modelo

No escopo do modelo, temos três aspectos que podem gerar limitações: as técnicas de processamento de linguagem natural, a abordagem de transformação escolhida e a granularidade do modelo de grafos (quais os artefatos de software que possuem representação no modelo). Além disso, a arquitetura do software no qual o MITRAS fará transformações também traz algumas implicações que serão discutidas.

5.1.1 Limitações de PLN

Quando trabalhamos com PLN, dificilmente é possível realizar uma interpretação sintática e semântica garantidamente correta. Devido a isso, alguns *parsers* trabalham com abordagens probabilísticas (TOUTANOVA; MANNING, 2000). O fato de mesmo pessoas fluentes em um idioma precisarem de contexto para minimizar a ambiguidade de algumas frases (RUSSELL et al., 2013) nos leva a crer que as limitações de *parsers* automatizados não vão acabar tão cedo.

Alguns subproblemas no espectro de PLN atingem margens de acerto bastante positivas quando testados contra corpus de texto, a quem defenda que esse tipo de problema já está resolvido, como é o caso do POS Tagging com índices superiores a 97% ((MANNING, 2011)). Embora resultados como esse sejam bastante animadores, devemos lembrar que uma ferramenta construída sobre um resultado que é correto em 97% dos casos, não vai conseguir superar essa acurácia, e se for aplicado um segundo processo que também gera um grau de incerteza esses erros acabam por se propagar.

A avaliação da acurácia de ferramentas de processamento de linguagem natural normal-

mente é realizada frente a um corpus de texto, para o idioma inglês existem bons corpus de texto já previamente classificado para avaliação (SANTORINI, 1990). Contudo, para problemas em domínios de conhecimento mais específicos, fica difícil encontrar um corpus de texto para que essa avaliação seja sequer realizada. No caso do MITRAS, seria ideal ter um banco de solicitações em linguagem natural sobre modificações no software. O que temos semelhante a isso são os logs de repositório *opensource* no qual *tickets* são avaliados em conjunto por programadores antes de serem encaminhados para desenvolvimento, esse tipo de histórico de conversa é diferente de uma solicitação hipotética que seria encaminhadas para o MITRAS, portanto não é ideal para ser usado nesse tipo de avaliação.

A alternativa nesse caso é criar um corpus sintético e realizar avaliação do módulo de PLN em laboratório. Com o sistema já em uso, daí sim seria possível coletar as solicitações e fazer um processo de curadoria nas solicitações não identificadas pelo MITRAS, assim seria possível criar um corpus de solicitação depois que o sistema estivesse em uso. Na prática, essa é a forma como esta limitação é tratada no projeto do MITRAS, através da disponibilização experimental do protótipo pela web e contínua coleta das solicitações de modificações para a criação de um corpus de solicitações de alterações de software.

5.1.2 Limitações do Modelo de Grafos

Conforme já foi detalhado anteriormente, o modelo de Grafos implementado pelo MITRAS possui uma série de restrições para torná-lo mais abstrato. Dessa forma, existem elementos do código fonte, como classes e associações, que possuem vértices e arestas correspondentes no grafo e outros elementos, como o código dos métodos, que não possuem correspondência. A consequência dessa abstração é tornar o sistema menos expressivo, impossibilitando que alguns tipos de modificação do software sejam mapeados em transformações.

Para ilustrar essas consequências suponhamos uma abstração ao extremo, na qual todo o sistema é representado por apenas um vértice no grafo, nesse caso o modelo de grafos gerado é extremamente simples, porém pouco se pode fazer com esse vértice a não ser excluí-lo, de modo que tal nível de abstração não tenha utilidade prática. No outro extremo, temos um grafo que captura todas as minuciosidades do código fonte, nesse caso cada linha de código pode ser identificada em um ou mais nodos e todas suas interações estão mapeadas por arestas, essa abordagem detalhista possibilita que qualquer alteração seja realizada via transformações, porém devido ao modelo possuir uma equivalência direta com o código fonte, poderíamos simplesmente descartar o modelo e ficar apenas com o código fonte, algo que tornaria questionáveis os benefícios de realizarmos alterações no modelo e não diretamente em código.

Para entender melhor quais os tipos de alterações que o modelo proposto abre mão de atender devido a seu nível de detalhamento intermediário (ver 3), nos próximos parágrafos será apresentada uma avaliação realizada no repositório de código e registro de *issues* do OpenMRS. Esses registros mantêm informação livre e gratuita sobre as alterações realizadas no sistema, já

devidamente classificadas e vinculadas às submissões de código fonte, bem como as discussões entre desenvolvedores durante o processo de desenvolvimento.

Foram avaliadas as *issues* com *issue type* igual a *New Feature* com status de encerrada e criadas a partir do começo de 2015 no projeto *openmrs-core*, os dados foram exportados do sistema online de *issues* do OpenMRS (OPENMRS ISSUES, 2013) avaliados localmente. Ao aplicar esses filtros, foram localizados 57 registros de *issues* com discussão de desenvolvedores e *commits* de código fonte devidamente associados.

Cada um desses registros foi avaliado e classificado como "Apto"nos casos em que o modelo de Grafos do MITRAS é suficientemente expressivo para tratar a alteração ou "Não Apto"nos casos contrários. É importante ressaltar aqui que as transformações de todos os casos "Aptos"não foram codificadas, o que foi realizado foi uma avaliação da viabilidade de aplicar uma transformação para realizar a alteração. Essa avaliação foi realizada principalmente analisando as mudanças de código fonte dos *commits* anexados as *issues*, alterações que trabalharam apenas com interfaces de classes/métodos, implementação de novas classes/métodos e integração de métodos já existentes foram classificadas como "Apto", alterações que envolvem mudança de algoritmos em métodos previamente existentes foram classificadas como "Não Apto".

A Figura 29 exibe um resumo gráfico das *issues* classificadas. Do total de 57, 12 estariam aptas para tratamento com a versão atual do modelo de grafos proposto, isso representa que 21% das alterações realizadas nesse projeto poderia em princípio ser tratada por alguma ferramenta de transformação com mesmo nível de abstração utilizado pelo MITRAS.

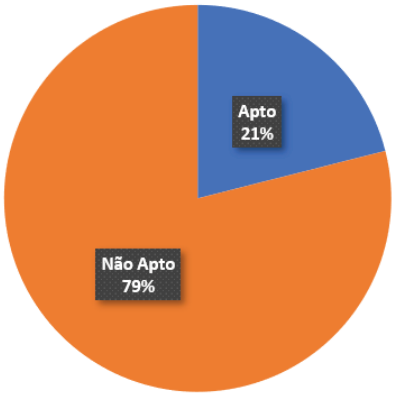

Figura 29 – Resumo de Issues Classificadas

A primeira vista o resultado de 21% aparenta ser baixo, contudo precisamos atentar que o repositório analisado realiza o versionamento do *core* do sistema, um local tipicamente reservado para alterações críticas e que devem ser válidas para todos os usuários, independentemente de suas necessidades mais específicas. Customizações e outras alterações consideradas como manutenções perfectivas, que são o foco do MITRAS, não entram nesse tipo de repositório. Dito isso, 21% é considerado um resultado satisfatório, visto que não é objetivo do projeto realizar todas as atividades de manutenção, mas apenas tornar viável que algumas delas sejam realizadas de forma automatizada.

Para encerrar essa análise, vamos mostrar uma transformação que poderia ser aplicada para resolver a solicitação da *issue* de id 91607 do OpenMRS, essa *issue* foi classificada como "Apto"e foi escolhida de forma arbitrária.

```
924         public boolean isPatient() {                    924         public boolean isPatient() {
925                 return isPatient;                       925                 return isPatient;
926         }                                               926         }
927                                                         927
                                                            928 +       // will replace isPatient in 2.x, see
                                                                https://issues.openmrs.org/browse/TRUNK-5161
                                                            929 +       public boolean getIsPatient() {
                                                            930 +               return isPatient;
                                                            931 +       }
                                                            932 +
928         /**                                             933         /**
```

Figura 30 – Comparação lado a lado do commit realizado no tratamento da *issue* 91607

A Imagem 30 mostra a alteração realizada no código fonte para resolver essa *issue*. Esse *diff* foi realizado no arquivo *Person.java*, no qual pode ser verificado a esquerda da Imagem o código fonte antes da alteração e a direita o código fonte após a alteração, após a adição do método *getIsPatient*.

Como essa alteração envolve a criação de um método do tipo *getter*, e não envolve alteração no corpo de um método específico, o modelo de grafos definido para o MITRAS é suficientemente expressivo para tratar essa alteração. No modelo proposto, métodos são representados por vértices com tipagem de *method* e ficam relacionados com a classe via uma aresta de mesmo tipo.

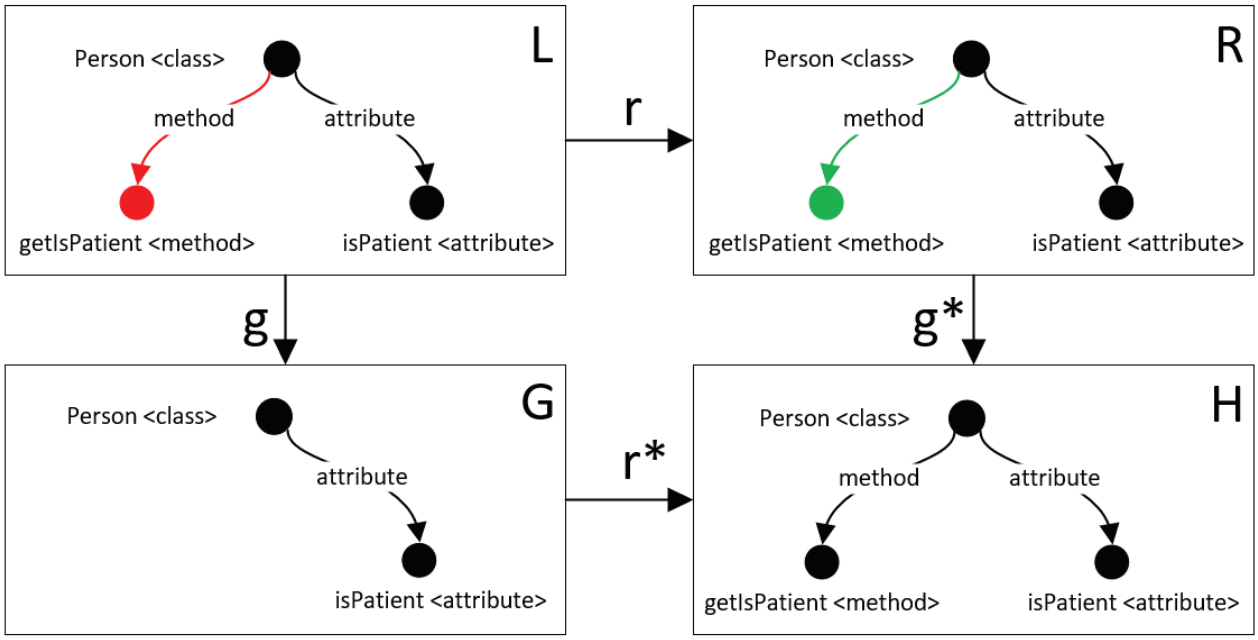

Figura 31 – Transformação para tratamento da *Issue* 91607

Na Imagem 31 podemos identificar todos os elementos necessários para aplicação de uma transformação que realize o tratamento da *issue* 91607 do OpenMRS. No grafo L a NAC, realçada em vermelho, indica a necessidade de que o método *getIsPatient* não exista para que essa transformação seja aplicada. No grafo do lado direito R, o vértice e a aresta em verde devem ser criados após a aplicação da regra r. Os grafos G e H mostram os estados anteriores e posteriores a transformação, respectivamente (por questões de simplicidade, os vértices e arestas não identificados pelos morfismos g e $g*$ não foram representados na figura, em uma situação prática, G e H representam o grafo inteiro do sistema antes e após a transformação).

Ao acessar as discussões da *issue* em questão, percebemos que essa alteração foi realizada em três versões diferentes do sistema, porém analisando os três *commits* diferentes, podemos perceber que apenas a transformação detalhada na Figura 31 é suficiente para as diferentes versões do sistema. Isso ocorre pois nessas situações foi possível estabelecer o morfismo g do grafo L para o grafo abstrato do sistema e satisfazer a NAC simultaneamente em todas as versões alteradas, sendo essas as condições necessárias para a aplicação da transformação.

As onze demais *issues* também admitem tratamento via transformações de grafo usando os mesmos princípios detalhados na seção 2.5 de forma intuitiva, na seção 4.4 de forma formal e na seção 4.2.5 com exemplos, porém as transformações de cada uma dessas *issues* não serão apresentadas.

Para os 45 casos considerados como "Não Apto", não foi possível criar transformações no modelo proposto que envolvessem todas as alterações com *commit* no repositório. Nesses casos, os motivos que levaram a inviabilidade foram diversos, porém os casos que mais apareceram foram, alterações de código dentro de métodos pré-existentes e modificação de arquivos de teste unitário.

5.1.3 Limitações das Transformações de Grafos

O modelo de grafos definido é manipulado utilizando transformações de grafo, dessa forma toda a modificação que será realizada no modelo (e por consequência no sistema) deve ser antes representada por uma transformação de grafos usando a abordagem algébrica. Na prática isso nos impõe a limitação de conseguir representar apenas modificações que possam ser identificadas usando morfismos de grafo, regras de criação de vértices e arestas e a NAC, qualquer alteração que não admita representação usando essas ferramentas não pode ser sintetizada a partir do MITRAS.

Nesse caso, apesar de nossa intuição levar a crer que o mecanismo de transformação insere limitações no processo, na prática isso não é verdade. É importante salientar que do ponto de vista teórico a abordagem algébrica para transformação de grafos cria sistemas computacionalmente completos (equivalentes a uma Máquina de Turing)(HABEL; PLUMP, 2001). Dessa forma, qualquer computação realizada pode ser simulada usando um sistema de transformação de grafo, assim não perdemos poder de expressão devido a esse formalismo.

Um aspecto negativo de trabalhar com um sistema computacionalmente completo para transformar o modelo de grafos é que incorremos nas mesmas limitações de uma Máquina de Turing, por exemplo, identificar que o critério de parada de uma transformação é atendido não é possível visto que isso seria equivalente a resolver o problema da parada de uma máquina de computação universal, problema reconhecidamente não computável.

5.1.4 Questões Arquiteturais

A primeira etapa para aplicação de uma transformação de grafo é encontrar um morfismo entre o grafo definido na transformação (do lado esquerdo) e o grafo abstrato do sistema. Encontrar um desses morfismos significa achar um local potencial do sistema que aceita esse tipo de transformação.

A transformação 1 (ver 4.2.5) tem como objetivo adicionar novos campos em cadastros do sistema. Essa transformação idealmente deveria ser aplicável a todos os cadastros do sistema, mas na prática isso não ocorre. A imagem 32 mostra o caso de um cadastro com implementação diferente que não satisfaz o morfismo com o grafo do lado esquerdo apresentado na imagem 16, o campo em vermelho é destacado pois ele fica vinculado diretamente ao formulário e não a um painel.

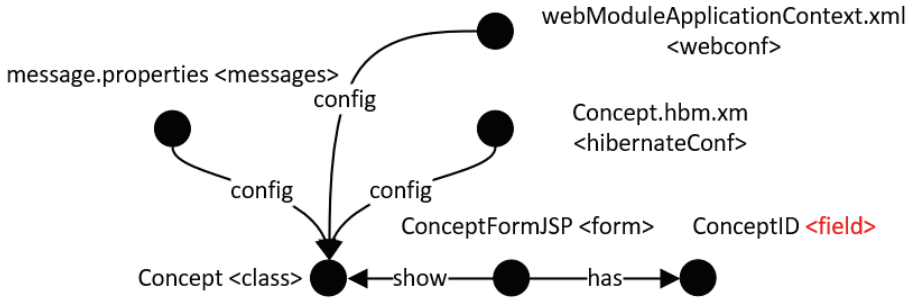

Figura 32 – Cadastro de *Concepts* não identificado pela Transformação 1

Nesse caso é possível perceber uma modificação na arquitetura do cadastro. Essa versão, não identificada pelo morfismo, implementa um cadastro sem classes intermediárias, e organiza os campos (destacado em vermelho) diretamente no formulário e não em um painel como os demais. Para atender a cadastros que usem esse tipo de implementação, seria necessário desenvolver uma variação da transformação 1, com grafos e morfismos adequados.

A avaliação desse caso específico nos mostra que existem limitações do MITRAS que não são inerentes ao próprio MITRAS, mas ao sistema ao qual as transformações são aplicadas. Um sistema que trabalha com uma arquitetura bem definida, no qual existe reuso de código e que padrões de projeto são implementados, as transformações tendem a ser mais genéricas, ou seja, se aplicam a mais lugares do sistema. De forma contrária, em sistemas que não possuem padronização, nos quais cada parte é desenvolvida com uma arquitetura única, as transformações tendem a perder generalidade e possibilitar aplicação apenas em poucos lugares.

Porém, mesmo nesta situação ainda é perfeitamente possível utilizar as transformações de grafos para codificar o conhecimento de como fazer modificações e manutenções no sistema, apenas necessitando de um maior número dessas transformações.

5.2 Limitações do Protótipo

A aplicação desenvolvida como prova de conceito possui algumas limitações. Não estamos interessados aqui em discorrer sobre limitações que todos sistemas computacionais reais possuem, como limites de processamento e/ou memória, mas sim nas simplificações que foram realizadas para construir o protótipo e realizar os testes.

Na etapa de extração de modelo, idealmente seria interessante possuir um *parser* completo da linguagem java, bem como *parsers* para arquivos html e xml. Na prática para construção do modelo abstrato de grafos, foram utilizadas ferramentas de análise de código fonte que geram modelos intermediários em XML, esses arquivos intermediários então foram avaliados para a criação do modelo abstrato de grafos.

Referente a construção da ontologia de interface, devido a necessidade de realizar uma execução assistida do sistema para descobrir vínculos entre conceitos, interfaces de usuário e entidades, as ontologias do protótipo não cobrem o sistema inteiro, mas apenas algumas telas administrativas que foram utilizadas nos experimentos.

O protótipo também conta com um conjunto de regras reduzido para identificação de frases e sinônimos. Aumentando o numero de léxicos ou criando novas regras seria possível identificar novos tipos de frase e enriquecer as capacidades de dialogo do MITRAS com o usuário.

Quanto ao banco de transformações, conforme detalhado em 4.2.5, é possível solicitar ao sistema quatro tipos de transformações diferentes, essas transformações são validas em diversos locais do OpenMRS, porém para um trabalho futuro seria interessante explorar mais opções de transformações.

6 METODOLOGIA EXPERIMENTAL

Com intuito de atingir os objetivos da dissertação, o protótipo foi submetido a testes em ambiente controlado, comparando alterações realizadas pelo MITRAS com alterações realizadas por desenvolvedores.

Nesse experimento, usuários interagiram com o MITRAS em linguagem natural de modo a realizar algumas alterações no sistema conforme um roteiro pré-definido. Em outro momento, foi solicitado para que um grupo de desenvolvedores de perfil júnior seguir o mesmo roteiro, codificando as alterações solicitadas sem acesso ao MITRAS.

Durante o experimento, a área de trabalho foi gravada de modo que todas as interações fiquem registradas para análise posterior. Posteriormente, os registros de ambos tipos de sujeitos de teste foram analisados para averiguar quais as atividades do roteiro foram completadas, que tipo de alterações foram realizadas no código fonte e quanto tempo cada sujeito de teste levou para fazer as alterações.

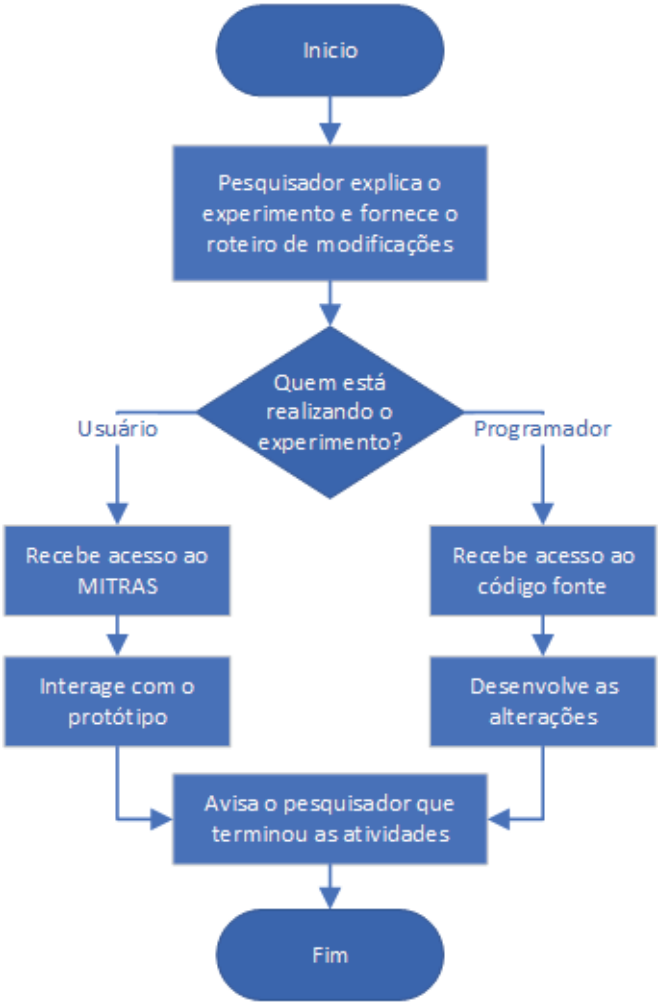

Figura 33 – Fluxograma do Experimento

O fluxograma da Figura 33 detalha o procedimento que foi realizado na experimentação. É

possível perceber que o roteiro é bastante semelhante para usuários e para programadores, tendo como principal diferença a ferramenta a qual eles terão acesso. O MITRAS foi disponibilizado para usuários e o código fonte para programadores.

A etapa inicial do experimento, na qual o pesquisador explicou o funcionamento do processo e forneceu o roteiro de modificações é muito semelhante para programadores e usuários. A diferença é que para os programadores é incluído uma pequena explicação sobre a estrutura do código fonte do OpenMRS enquanto que para usuários é realizada uma explicação sobre o funcionamento do MITRAS.

Para garantir que tanto usuários como programadores recebam acesso a um ambiente equivalente, foi montada uma máquina virtual com todas as ferramentas necessárias para o experimento: OpenMRS, código fonte, MITRAS e IDE de desenvolvimento JAVA (Eclipse). Ao término do experimento foram coletados logs de execução do MITRAS para os usuários e o código fonte para os desenvolvedores.

As três atividades solicitadas no roteiro do experimento estão transcritas a seguir:

1. Adicionar informação para CPF nas informações do paciente.

2. Ocultar os campos de latitude e longitude do endereço do paciente

3. Inverter a ordem dos campos Province e Division

As atividades que foram indicadas no roteiro estão relacionadas diretamente com as transformações implementadas no protótipo do MITRAS. De modo que cada uma das três atividades demanda o uso de uma transformação diferente e coube ao usuário solicitar essas mudanças para o sistema e ao programador descobrir como codificar essas solicitações.

Juntamente com essas solicitações foram disponibilizadas cópias de tela do OpenMRS dos elementos de interface a qual cada uma das solicitações dizia respeito. Para os usuários, estava também disponível um vídeo com um tutorial de uso do MITRAS, caso o usuário tivesse dúvidas durante o uso do sistema poderia assistir o vídeo com uma versão gravada da explicação dada antes do início do experimento.

6.1 Experimento com Usuários

A tabela 6 mostra um resumo das pessoas que realizaram o experimento, nela podemos perceber que o tempo de experiência com desenvolvimento do grupo de programadores é menor do que três anos, característica de desenvolvedores com perfil júnior.

Cabe salientar que todas as pessoas não realizaram o experimento simultaneamente, todos os participantes trabalharam individualmente em suas máquinas. Houveram quatro usuários que assistiram a explicação sobre o experimento em grupos de dois, porém mesmo nesses casos a interação com o MITRAS foi individual.

Tabela 6 – Dados dos participantes no experimento

Participante	Idade	Tempo Experiência
Usuário 1	37	-
Usuário 2	28	-
Usuário 3	32	-
Usuário 4	48	-
Usuário 5	25	-
Usuário 6	26	-
Usuário 7	22	-
Usuário 8	46	-
Programador 1	18	1 ano
Programador 2	21	<1 ano
Programador 3	22	2 anos
Programador 4	21	<1 ano

A figura 34 é um recorte das interações que um usuário realizou com o MITRAS para desenvolver a tarefa 1 do experimento. Nessa atividade, a primeira interação do usuário não forneceu o dado do local onde o novo campo deveria ser disponibilizado, o MITRAS avisou ao usuário que por sua vez encaminhou nova solicitação com as informações faltantes.

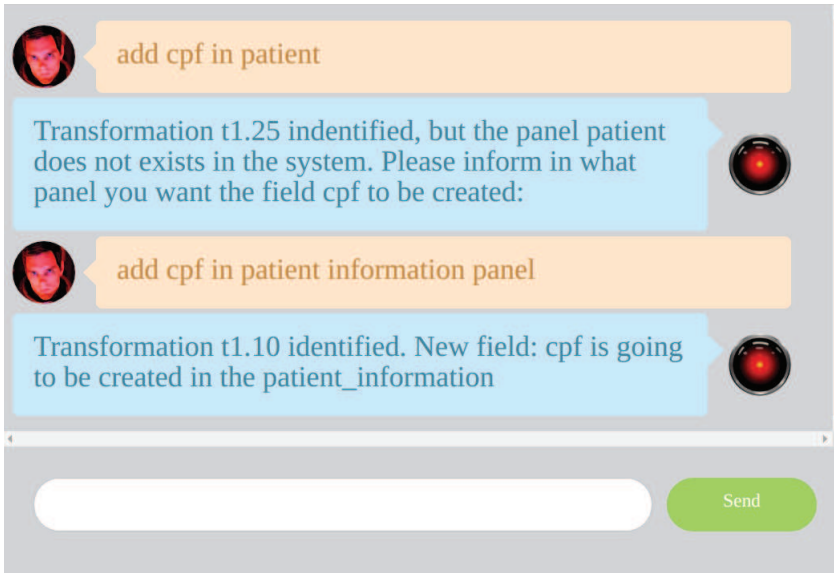

Figura 34 – Interação realizada por um dos usuários para realizar a tarefa 1

Os oito usuários realizaram em seu total 59 interações com o MITRAS, para atingir seus objetivos e realizar as alterações propostas, uma média de 7,4 interações por usuário, sendo no mínimo quatro e no máximo onze interações durante o experimento. O usuário mais rápido levou 5 minutos para realizar as alterações enquanto o usuário que demorou mais tempo levou 14 minutos interagindo com o sistema. A tabela 7 resume essas informações.

Em ambas tabelas, 7 e 9, é possível verificar o resultado de cada uma das três solicitações de modificação, o resultado pode ser: Concluído, quando a solicitação foi plenamente atin-

Tabela 7 – Resumo dos resultados dos usuários

Usuário	Atividade 1		Atividade 2		Atividade 3		Total	
	Frases	Status	Frases	Status	Frases	Status	Frases	Tempo
1	1	Concluída	2	Concluída	2	Concluída	6	5 min
2	2	Concluída	3	Concluída	3	Concluída	8	6 min
3	2	Concluída	5	Concluída	4	Parcialmente	11	14 min
4	1	Concluída	2	Concluída	4	Concluída	7	6 min
5	2	Concluída	4	Concluída	4	Parcialmente	10	12 min
6	2	Concluída	3	Concluída	3	Concluída	8	10 min
7	1	Concluída	3	Concluída	2	Concluída	6	6 min
8	1	Concluída	2	Concluída	1	Concluída	4	5 min

Tabela 8 – Resultado PLN por atividade

Atividade #	Identificado	Não Identificado	Total
1	10	2	12
2	16	8	24
3	15	8	23
Total	41	18	59

gida; Parcialmente Concluído, quando a solicitação foi concluída parcialmente ou quando as modificações geraram algum efeito colateral indesejado; Não Realizado, quando a alteração não foi implementada ou foi implementada errada. É possível perceber que nesse quesito tanto programadores quanto usuários atingiram resultados semelhantes.

Cabe destacar aqui um *feedback* dado informalmente por um usuário após a realização do experimento. O usuário questionou o pesquisador o que aconteceria caso o MITRAS gerasse alguma inconsistência no sistema - "um programador teria que ser acionado para resolver ou o MITRAS teria a capacidade de se corrigir?". Isso mostra que o usuário se sentiu um tanto apreensivo por poder interagir com o sistema de uma forma que ainda não estava habituado.

A tabela 8 detalha o que ocorreu com cada uma das solicitações em linguagem natural processadas pelo MITRAS durante o experimento. A coluna "Identificado"mostra a quantidade de solicitações que identificaram a transformação solicitada, a coluna "Não Identificada"mostra o total de solicitações que não identificaram diretamente uma transformação. Dentre essas 18 frases as quais o agente PLN não identificou uma transformação, temos que: 2 solicitações de operações que o MITRAS não faz; 2 erros de ortografia na língua inglesa; 9 frases que falharam na avaliação sintática ou semântica; 5 problemas de identificação de elementos de interface na ontologia de sinônimos. Se removermos as duas questões cujos usuários cometeram erros de inglês, temos que no escopo do experimento 72%[1] das interações realizadas pelo usuário foram identificadas como uma transformação, e temos que a acurácia do agente de PLN foi de 84,7%[2].

[1]Calculado por $Id/(Id + NId - EI)$ onde Id é o número de solicitações identificadas, NId é o número de solicitações não identificadas e EI é o número de questões com erro de ortografia.

[2]Considerando que a acurácia é medida por: $(NInt - ESS)/NInt$ onde: $NInt$ é o número total de interações

Tabela 9 – Resumo dos resultados dos programadores

Programador	Atividade 1	Atividade 2	Atividade 3	Tempo
1	Concluída	Concluída	Concluída	35 min
2	Parcialmente	Concluída	Concluída	23 min
3	Concluída	Concluída	Concluída	22 min
4	Concluída	Concluída	Concluída	25 min

Os usuários 3 e 5 tiveram indicação de conclusão parcial na Atividade 3 por motivos diferentes. O usuário 3 solicitou a movimentação dos campos para posições diferentes daquelas constantes no roteiro do experimento, já o usuário 5 moveu mais campos do que o solicitado. Em ambos os casos não fica claro se foi problema de compreensão do experimento por parte dos usuários ou se eles não souberam como se expressar nas interações com o MITRAS, porém as alterações implementadas ficaram de acordo com as frases digitadas no experimento.

6.2 Experimento com Programadores

No grupo de desenvolvedores que realizaram o experimento, podemos identificar que a grande maioria das atividades também foi totalmente concluída. O programador 2 não encontrou as classes de acesso a dados para implementar a persistência, portanto deixou a atividade 1 parcialmente desenvolvida.

O programador mais rápido para realizar as alterações demorou 22 minutos enquanto o programador que demorou mais tempo realizou o experimento em 35 minutos. Na tabela 9 é possível verificar diretamente o tempo gasto nas alterações por cada um dos programadores.

Após acompanhar os programadores, percebeu-se que boa parte do tempo foi utilizado para ler e compreender a aplicação, pois nenhum deles possuía familiaridade com o código fonte do OpenMRS. Apenas após uma leitura inicial do código que os desenvolvedores começaram suas atividades, baseadas em procurar o código antigo e replicar ele para atender as solicitações.

Foi possível perceber também diferenças entre os códigos escritos pelos desenvolvedores. Três desenvolvedores preferiram manter os padrões para nomeação de atributos e métodos já utilizados nas classes que demandaram alteração, porém um deles achou melhor usar nomes e codificar de uma forma diferente daquilo que já era praticado no OpenMRS. A figura 35 ilustra uma dessas diferenças entre os códigos fontes desenvolvidos.

6.3 Comparação entre Programadores e Usuários

Avaliando os resultados de programadores e usuários, a primeira questão que fica evidente é que as dificuldades surgiram em atividades diferentes, enquanto dois usuários executaram a

do experimento e ESS é o número de erros sintáticos ou semânticos ao avaliar a frase.

```
String gender = person.get(GENDER);
String cpf = person.get(CPF);
String name = person.get(NAME);
String birthdate = person.get(BIRTH_DATE);
String age = person.get(AGE);

log.debug("name: " + name + " birthdate: " + birthdate + " age: " + age + " gender: " + gender);

if (StringUtils.isNotEmpty(name) || StringUtils.isNotEmpty(birthdate) || StringUtils.isNotEmpty(age)
        || StringUtils.isNotEmpty(gender)|| StringUtils.isNotEmpty(cpf)) {
```

```
String gender = person.get(GENDER);
String name = person.get(NAME);
String birthdate = person.get(BIRTH_DATE);
String age = person.get(AGE);
String codCPF = person.get("codCPF");

log.debug("name: " + name + " birthdate: " + birthdate + " age: " + age + " gender: " + gender);

if (StringUtils.isNotEmpty(name) || StringUtils.isNotEmpty(birthdate) || StringUtils.isNotEmpty(age)
        || StringUtils.isNotEmpty(gender)|| StringUtils.isNotEmpty(codCPF)) {
```

Figura 35 – Comparação de código entre dois desenvolvedores

atividade 3 com sucesso parcial, um desenvolvedor apresentou resultado parcial na atividade 1. A dificuldade de um usuário de expressar uma modificação em linguagem natural não parece indicar necessariamente uma dificuldade maior no desenvolvimento.

Quanto ao tempo de envolvimento nas atividades, os usuários foram mais rápidos, apresentando tempo médio de 8 minutos e 48 segundos contra 26 minutos e 30 segundos para os programadores. Isso ocorre, pois, a atividade de compreensão do código fonte foi abstraídas nas transformações de grafo, de modo que o usuário não precisa compreender a estrutura do código fonte para realizar as alterações enquanto o desenvolvedor precisa ler e ter um mínimo entendimento para poder prosseguir com as alterações.

Ambos os grupos apresentaram participantes que não conseguiram realizar o tratamento total de todas as atividades, no caso dos usuários houve modificações realizadas diferente daquilo que estava sendo proposto enquanto que um programador não conseguiu finalizar a alteração. No caso do programador interpretamos que esse é um problema que aparece apenas em uma situação de experimentação, pois em um desenvolvimento real ele não teria "desistido"da modificação tão rápido, ele poderia ter acionado colegas ou investido mais tempo na leitura do código fonte. Já no caso dos usuários, é possível que, ao perceber uma solicitação de modificação de posição de campos eles se deram satisfeitos ao conseguir mudar a posição deles sem dar o devido rigor as posições solicitadas ou a possíveis efeitos colaterais. Ambos os grupos apresentaram resultados positivos em 92% das alterações propostas.

Outra questão que merece destaque é que as alterações realizadas no código fonte no contexto do experimento com usuários foi igual, isso ocorre pois o agente PLN detecta as diferentes escritas dos usuários e extrai as mesmas informações para o agente de transformações trabalhar, de modo que a faze de transformação de grafos e de injeção de código fosse igual para todos os usuários. No caso dos desenvolvedores isso não ocorre, como cada programador trabalha

de uma forma ligeiramente diferente, é natural que o mesmo problema seja resolvido usando abordagens diferenciadas.

6.4 Análise de Ameaças

O experimento proposto nessa dissertação comparou os resultados de usuários do MITRAS e programadores dentro do escopo de um roteiro com atividades pré-determinadas pelo pesquisador. Os resultados foram apresentados nas seções anteriores, aqui será feita uma discussão dos fatores que foram detectados como ameaça para o experimento.

6.4.1 Vocabulário Limitado

A interação dos usuários com o MITRAS se dá através de uma interface de linguagem natural, ou seja, os usuários podem interagir com o protótipo a partir de frases abertas no idioma inglês. Devido a essa flexibilidade, é natural de se esperar que cada usuário vá se referir a elementos da interface gráfica de uma forma diferente, o painel onde ficam armazenados os nomes do usuário pode ser referido por *"Name Panel", "Patient Names", "Names Record", "Name File"* ou *"Name Data"* por exemplo.

Essas diversas formas de se referir a elementos do sistema são difíceis de prever em tempo de desenvolvimento. De modo que é possível (conforme foi verificado no experimento) que alguns usuários usem palavras ou expressões que não estão disponíveis nas ontologias do MITRAS e, por conseguinte, sua frase não será compreendida ou será erroneamente classificada.

Essa ameaça é mitigada devido aos vocabulários do agente NLP do MITRAS estarem disponíveis em ontologias que podem ser expandidas com novos termos ou com sinônimos de termos previamente existentes. Esse processo para incremento das ontologias poderia ser feito manualmente em um primeiro momento, através de uma curadoria das frases enviadas ao MITRAS com termos que não foram identificados na ontologia, ou em um segundo momento até mesmo de forma automatizada pelo próprio chat, no qual o usuário poderia informar ao MITRAS como são os nomes de elementos do sistema.

6.4.2 Inserção de Manutenções

Conforme o *feedback* de um dos usuários, devemos discutir um pouco a possibilidade de o processo de transformação do MITRAS gerar bugs no sistema.

De antemão podemos dizer que o MITRAS não está habilitado para executar manutenções corretivas, ou seja, problemas existentes do sistema não serão tratados pelas transformações propostas, esse tipo de manutenção deverá seguir o procedimento tradicional no qual um programador avalia o código fonte e faz as alterações necessárias.

Agora vamos supor que o MITRAS gere um novo problema ao aplicar uma transformação

no sistema, existem duas etapas nas quais o MITRAS trabalha com o software, a etapa de transformação na qual o software é inspecionado em nível de modelo e a etapa de injeção na qual o software é inspecionado em nível de código fonte.

Caso a transformação tenha sido mal projetada, o MITRAS pode vir a detectar que uma transformação pode ser aplicada em um lugar errôneo, já se um *snippet* de código foi escrito erroneamente o programa pode deixar de compilar. No caso específico do OpenMRS, existem uma série de testes automatizados que detectam esse tipo de alteração, dando maior segurança que esse tipo de problema será evitado na maioria dos casos. Assim, a versão do sistema não será atualizada devido a falha nos testes automatizados e o usuário pode solicitar que as transformações sejam desfeitas.

Outra questão interessante nesse aspecto é que, devido ao MITRAS ter sua etapa de transformações formalmente definida, teoricamente seria possível realizar provas de corretude de transformações, para assegurar que a transformação atinge o objetivo proposto e não causa efeitos colaterais. A etapa de injeção ainda não está formalizada, mas caso ela seja formalizada, seria possível realizar provas usando a árvore de parse da linguagem para garantir que a injeção gera um código passível de compilação. Ambas provas estão fora do escopo dessa dissertação mas ficam como ideia para trabalhos futuros.

6.4.3 Programadores Inexperientes

O grupo de programadores que realizou o experimento foi composto de estudantes e programadores com pouca experiência. Poderia ser argumentado que um grupo de programadores mais experientes ou que já tivesse familiaridade com o código fonte poderia realizar as alterações mais rápido.

De fato, programadores que conhecem um determinado projeto possuem mais facilidade em realizar alterações. Contudo, desenvolvedores mais experientes normalmente estão engajados em atividades mais complexas de desenvolvimento nas quais sua experiencia e conhecimento dos fontes são mais necessárias. Atividades mais simples, como as realizadas pelo MITRAS, normalmente ficam a cargo de profissionais menos experientes ou ainda em treinamento.

Assim sendo, lembrando que o objetivo do MITRAS não é atacar todo o tipo de alteração mas apenas manutenções evolutivas ou perfectivas mais simples, faz sentido comparar os resultados de usuários do MITRAS contra programadores com pouca ou nenhuma experiência de desenvolvimento, visto que parece ser razoável supor que esse tipo de programador trabalharia com as tarefas que o MITRAS realiza automação.

7 CONSIDERAÇÕES FINAIS

Essa dissertação apresentou um modelo de transformação de software cujo objetivo é auxiliar no processo de manutenção de sistemas, oferecendo autonomia a usuários finais de sistemas no que tange manutenções perfectivas e adaptativas. O modelo foi elaborado com base em formalismos algébricos para transformação de grafos, promovendo um nível de abstração suficientemente expressivo para promover modificações no sistema, porém abstrato o suficiente para não expor todos seus detalhes de implementação.

Outros esforços para transformações de sistemas se afastam da proposta do MITRAS seja por trabalharem mais em baixo nível, com noções de programação automatizada e exploração de grandes espaços de busca derivados de cadeias possivelmente infinitas de programas válidos ou ainda por trabalharem com *toy problems*, deixando no ar o questionamento sobre sua aplicabilidade em sistemas reais. No caso do MITRAS, se optou desde o inicio pela sua avaliação e experimentação com um sistema real.

Dados coletados da literatura e de profissionais da indústria apresentam evidência inicial que manutenções perfectivas e adaptativas demandam esforço considerável no processo de desenvolvimento, mostrando a importância na pesquisa de novos métodos e tecnologias para amenizar essa carga de trabalho. Contudo a pesquisa realizada nessa dissertação foi limitada a região metropolitana de Porto Alegre, portanto seria necessário expandir a amostra em regiões diferentes para verificar se essa tendência se mantém.

Outro aspecto importante da dissertação é a proposta de um esquema de classificação para sistemas de transformação de software, a partir de uma classificação padronizada deve ser mais fácil comparar diferentes tipos de sistemas transformacionais e verificar sua complexidade, tempo de resposta e expressividade em termos de transformações.

Ao longo do trabalho o modelo foi aplicado a um sistema *open source* de grande porte e que está em uso real por uma variedade de instituições e usuários, no qual transformações para adição de informações persistentes, customizações de interface e criação de novas funcionalidades a partir de dados existentes foram realizadas. Um processo de mineração de repositório viabilizou a construção do modelo de grafos do sistema, a partir de tecnologias equivalentes a compiladores, porém com modificações para permitir uma representação intermediária adequada ao objetivo proposto.

Os dados coletados nos experimentos iniciais mostram que usuários conseguem realizar solicitações de mudanças para um sistema de transformações inteligente, dado que o sistema é capaz de reconhecer elementos de interface gráfica, suas relações e um conjunto de modificações pré-definidas em formato de transformações. Esses resultados foram atingidos sem realizar treinamento intensivo dos usuários mas apenas aproveitando seu conhecimento da interface do sistema.

Percebeu-se também que a dificuldade que programadores possuem no desenvolvimento não se transfere para os usuários do MITRAS. Enquanto um dos programadores teve dificul-

dade em encontrar todos os arquivos de código fonte que deveriam ser modificados, isso não foi problema para os usuários, visto que a localização dos arquivos fonte fica abstraída pelo MITRAS. Porém os usuários acabaram solicitando mais modificações do que foi solicitado, alterando a posição de campos sem necessidade.

De modo nenhum o tópico de modelos de transformação de sistemas foi exaurido durante essa pesquisa. De fato, ao longo do desenvolvimento foi possível perceber diversas possibilidades de novas pesquisas nesse âmbito, dentre as quais podemos destacar a criação automática da biblioteca de transformações a partir do histórico de *commits* realizados no repositório, exploração de formalismos diferenciados para transformações e técnicas diferenciadas para processamento de linguagem natural.

Sobre o módulo de processamento de linguagem natural, foram exploradas diversas ferramentas que formam o estado da arte para parse da língua inglesa (Parser de Stanford), aliado a técnicas baseadas na aplicação da lógica de primeira ordem para ontologias de termos do domínio de aplicação para inferência da intenção do usuário ao interagir com o sistema. Essa abordagem foi possível devido a restrição das interações em linguagem natural no MITRAS apenas ao domínio de solicitação de transformações de software e não em um cenário de diálogo mais aberto, no qual técnicas mais robustas deveriam ser aplicadas. Contudo, como não é objetivo dessa dissertação a criação de um sistema de conversa de propósito geral, as premissas e restrições impostas são coerentes com os objetivos da dissertação.

Como sugestão para o processo de interação com usuário final, o módulo de PLN poderia ser estendido para possibilitar tirar dúvidas sobre o sistema ou mesmo possibilitar a criação de novas transformações a partir da interação em linguagem natural.

Dessa forma, entende-se que a contribuição dessa dissertação é justamente o modelo MITRAS conforme foi proposto e sua primeira aplicação a um projeto *open source*, sugerindo sua viabilidade técnica. Apesar dos experimentos terem sido conduzidos com apenas um sistema, acredita-se que o modelo pode ser aplicado a uma gama considerável de projetos, visto que sua operacionalização se baseia em dois processos: extração de modelos a partir de código fonte e transformações de grafo que podem ser projetadas conforme a necessidade, intuitivamente ambos aparentam ser passíveis de automatização.

Essa primeira experimentação com o MITRAS permitiu que usuários executassem as mesmas alterações que desenvolvedores em um tempo menor e com qualidade de código semelhante dentro de um roteiro pré-estabelecido. Conforme a evolução do MITRAS, é possível que alguns desenvolvedores possam se ocupar projetando transformações genéricas ao invés de escrever apenas código fonte para uma modificação específica, assim os usuários poderiam escolher quais as transformações devem ser aplicadas em seu sistema.

REFERÊNCIAS

Ab. Rahim, L.; WHITTLE, J. A survey of approaches for verifying model transformations. **Software & Systems Modeling**, [S.l.], v. 14, n. 2, p. 1003–1028, may 2015.

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compilers**: principles, techniques, and tools. [S.l.]: Addison-wesley Reading, 2007. v. 2.

AMIRAT, A. et al. Refactor software architecture using graph transformation approach. In: INNOVATIVE COMPUTING TECHNOLOGY (INTECH), 2012 SECOND INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 117–122.

ANJORIN, A.; LEBLEBICI, E.; SCHÜRR, A. 20 years of triple graph grammars: a roadmap for future research. **Electronic Communications of the EASST**, [S.l.], v. 73, 2016.

AUSTIN, J. L. **Quando dizer é fazer**: palavras e ação. [S.l.]: Artes Médicas, 1990.

BAADER, F.; HORROCKS, I.; SATTLER, U. Chapter 3 Description Logics. **Foundations of Artificial Intelligence**, [S.l.], v. 3, 2008.

STAAB, S.; STUDER, R. (Ed.). Description logics. In: _____. **Handbook on ontologies**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 21–43.

BLACKBURN, P.; BENTHEM, J. F. A. K. v.; WOLTER, F. **Handbook of modal logic, volume 3 (studies in logic and practical reasoning)**. New York, NY, USA: Elsevier Science Inc., 2006.

BNF rules of java. Accessed: 2018-01-03, http://cui.unige.ch/isi/bnf/JAVA/AJAVA.html.

BONDY, J. A.; MURTY, U. S. R. et al. **Graph theory with applications**. [S.l.]: Citeseer, 1976. v. 290.

BOURQUE, P.; FAIRLEY, R. E. et al. **Guide to the software engineering body of knowledge (swebok (r))**: version 3.0. [S.l.]: IEEE Computer Society Press, 2014.

BRATMAN, M. E.; ISRAEL, D. J.; POLLACK, M. E. Plans and resource-bounded practical reasoning. **Computational intelligence**, [S.l.], v. 4, n. 3, p. 349–355, 1988.

BUCHMANN, I.; FRISCHBIER, S.; PUTZ, D. Towards an estimation model for software maintenance costs. In: SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), 2011 15TH EUROPEAN CONFERENCE ON, 2011. **Anais...** [S.l.: s.n.], 2011. p. 313–316.

CHEN, D.; MANNING, C. A fast and accurate dependency parser using neural networks. In: EMNLP), 2014., 2014. **Proceedings...** [S.l.: s.n.], 2014. p. 740–750.

CORDY, J. R. et al. Software engineering by source transformation-experience with txl. In: SOURCE CODE ANALYSIS AND MANIPULATION, 2001. PROCEEDINGS. FIRST IEEE INTERNATIONAL WORKSHOP ON, 2001. **Anais...** [S.l.: s.n.], 2001. p. 168–178.

CORMEN, T. H. **Introduction to algorithms**. [S.l.]: MIT press, 2009.

CORRADINI, A. et al. Handbook of graph grammars and computing by graph transformation. In: ROZENBERG, G. (Ed.). . River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997. p. 163–245.

DE MARNEFFE, M.-C. et al. Universal stanford dependencies: a cross-linguistic typology. In: LREC, 2014. **Anais...** [S.l.: s.n.], 2014. v. 14, p. 4585–4592.

DE ROOVER, C.; INOUE, K. The ekeko/x program transformation tool. In: SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), 2014 IEEE 14TH INTERNATIONAL WORKING CONFERENCE ON, 2014. **Anais...** [S.l.: s.n.], 2014. p. 53–58.

EHRIG, H. et al. **Fundamentals of algebraic graph transformation (monographs in theoretical computer science. an eatcs series)**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

EHRIG, H. et al. Modelling and static analysis of self-adaptive systems by graph transformation. In: **Graph and model transformation**. [S.l.]: Springer, 2015. p. 299–326.

EHRIG, H. et al. **Graph and model transformation**: general framework and applications. [S.l.]: Springer, 2015.

EHRIG, H.; KORFF, M.; LöWE, M. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In: INTERNATIONAL WORKSHOP ON GRAPH-GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 4., 1991, London, UK, UK. **Proceedings...** Springer-Verlag, 1991. p. 24–37.

EHRIG, H.; PFENDER, M.; SCHNEIDER, H. J. Graph-grammars: an algebraic approach. In: SWITCHING AND AUTOMATA THEORY, 1973. SWAT'08. IEEE CONFERENCE RECORD OF 14TH ANNUAL SYMPOSIUM ON, 1973. **Anais...** [S.l.: s.n.], 1973. p. 167–180.

ERLIKH, L. Leveraging legacy system dollars for e-business. **IT professional**, [S.l.], v. 2, n. 3, p. 17–23, 2000.

FINKEL, J. R.; GRENAGER, T.; MANNING, C. Incorporating non-local information into information extraction systems by gibbs sampling. In: OF THE 43RD ANNUAL MEETING ON ASSOCIATION FOR COMPUTATIONAL LINGUISTICS, 2005. **Proceedings...** [S.l.: s.n.], 2005. p. 363–370.

GAMMA, E. **Design patterns**: elements of reusable object-oriented software. [S.l.]: Pearson Education India, 1995.

GERSTING, J. L. **Mathematical structures for computer science**. [S.l.]: Macmillan Higher Education, 2014.

GIARETTA, P.; GUARINO, N. Ontologies and knowledge bases towards a terminological clarification. **Towards very large knowledge bases: knowledge building & knowledge sharing**, [S.l.], v. 25, p. 32, 1995.

GUARINO, N. Understanding, building and using ontologies. **International Journal of Human-Computer Studies**, [S.l.], v. 46, n. 2-3, p. 293–310, 1997.

HABEL, A.; PLUMP, D. Computational completeness of programming languages based on graph transformation. In: INTERNATIONAL CONFERENCE ON FOUNDATIONS OF SOFTWARE SCIENCE AND COMPUTATION STRUCTURES, 2001. **Anais...** [S.l.: s.n.], 2001. p. 230–245.

HAN, J. Mining structures from massive text data: will it help software engineering? In: AUTOMATED SOFTWARE ENGINEERING (ASE), 2017 32ND IEEE/ACM INTERNATIONAL CONFERENCE ON, 2017. **Anais...** [S.l.: s.n.], 2017. p. 2–2.

HOCHGESCHWENDER, N. et al. Graph-based software knowledge: storage and semantic querying of domain models for run-time adaptation. In: SIMULATION, MODELING, AND PROGRAMMING FOR AUTONOMOUS ROBOTS (SIMPAR), IEEE INTERNATIONAL CONFERENCE ON, 2016. **Anais...** [S.l.: s.n.], 2016. p. 83–90.

HOPCROFT, J. E. **Introduction to automata theory, languages, and computation**. [S.l.]: Pearson Education India, 2008.

IGWE, K.; PILLAY, N. Automatic programming using genetic programming. In: INFORMATION AND COMMUNICATION TECHNOLOGIES (WICT), 2013 THIRD WORLD CONGRESS ON, 2013. **Anais...** [S.l.: s.n.], 2013. p. 337–342.

ISO/IEC 12207 - Software life cycle processes. Geneva, CH: International Organization for Standardization, 2008. Standard.

ISO/IEC 14764 - Software Maintenance. Geneva, CH: International Organization for Standardization, 2006. Standard.

JACOBS, C.; GRUNE, D. Parsing techniques: a practical guide. **Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands**, [S.l.], 1990.

JILANI, A.; USMAN, M.; HALIM, Z. Model transformations in model driven architecture. **Universal Journal of Computer Science and Engineering Technology**, [S.l.], v. 1, 01 2010.

KAHANI, N. et al. Survey and classification of model transformation tools. **Software & Systems Modeling**, [S.l.], p. 1–37, 2018.

KATO, J.; IGARASHI, T.; GOTO, M. Programming with examples to develop data-intensive user interfaces. **Computer**, [S.l.], v. 49, n. 7, p. 34–42, 2016.

KINDLER, E.; WAGNER, R. **Triple graph grammars**: concepts, extensions, implementations, and application scenarios. [S.l.]: Tech. Rep. TR-ri-07-284, Department of Computer Science, University of Paderborn, Germany, 2007.

HONSELL, F.; MICULAN, M. (Ed.). Foundations for a graph-based approach to the specification of access control policies. In: _____. **Foundations of software science and computation structures**: 4th international conference, fossacs 2001 held as part of the joint european conferences on theory and practice of software, etaps 2001 genova, italy, april 2–6, 2001 proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 287–302.

LACK, S.; SOBOCINSKI, P. Adhesive categories. In: FOSSACS, 2004. **Anais...** [S.l.: s.n.], 2004. v. 2987, p. 273–288.

LENARDUZZI, V.; SILLITTI, A.; TAIBI, D. Analyzing forty years of software maintenance models. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING COMPANION, 39., 2017. **Proceedings...** [S.l.: s.n.], 2017. p. 146–148.

LöWE, M. Algebraic approach to single-pushout graph transformation. **Theoretical Computer Science**, [S.l.], v. 109, n. 1, p. 181 – 224, 1993.

MAMLIN, B. W. et al. Cooking up an open source emr for developing countries: openmrs–a recipe for successful collaboration. In: AMIA ANNUAL SYMPOSIUM PROCEEDINGS, 2006. **Anais...** [S.l.: s.n.], 2006. v. 2006, p. 529.

MANNING, C. D. Part-of-speech tagging from 97% to 100%: is it time for some linguistics? In: INTERNATIONAL CONFERENCE ON INTELLIGENT TEXT PROCESSING AND COMPUTATIONAL LINGUISTICS, 2011. **Anais...** [S.l.: s.n.], 2011. p. 171–189.

MANNING, C. D. et al. The Stanford CoreNLP natural language processing toolkit. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS (ACL) SYSTEM DEMONSTRATIONS, 2014. **Anais...** [S.l.: s.n.], 2014. p. 55–60.

MCBURNEY, D.; SLEEP, M. R. Graph rewriting as a computational model. In: UK/JAPAN WORKSHOP ON CONCURRENCY, 1989. **Anais...** [S.l.: s.n.], 1989. p. 235–256.

MCLAUGHLIN, L. Automated programming the next wave of developer power tools. **IEEE Software**, [S.l.], v. 23, n. 3, p. 91–93, 2006.

MEHNER-HEINDL, K.; MONGA, M.; TAENTZER, G. Analysis of aspect-oriented models using graph transformation systems. In: **Aspect-oriented requirements engineering**. [S.l.]: Springer, 2013. p. 243–270.

MUNN, K.; SMITH, B. **Applied ontology**: an introduction. [S.l.]: Walter de Gruyter, 2008. v. 9.

NIVRE, J. et al. Universal dependencies v1: a multilingual treebank collection. In: LREC, 2016. **Anais...** [S.l.: s.n.], 2016.

OPENMRS issues. Accessed: 2018-07-18, https://issues.openmrs.org/.

PADUELLI, M. M. **Manutenção de software**: problemas típicos e diretrizes para uma disciplina específica. 2007. Tese (Doutorado em Ciência da Computação) — Universidade de São Paulo, 2007.

PIGOSKI, T. M. **Practical software maintenance**: best practices for managing your software investment. [S.l.]: Wiley Publishing, 1996.

PRESSMAN, R. S. **Software engineering**: a practitioner's approach. [S.l.]: Palgrave Macmillan, 2005.

RADOSEVIC, D.; OREHOVACKI, T.; MAGDALENIC, I. Towards software autogeneration. In: MIPRO, 2012 PROCEEDINGS OF THE 35TH INTERNATIONAL CONVENTION, 2012. **Anais...** [S.l.: s.n.], 2012. p. 1076–1081.

RAMOS, M. V. M. Linguagens formais e autômatos. **Apostila, Universidade Federal do Vale do São Francisco, Curso de Engenharia de Computação**, [S.l.], 2008.

RAO, A. S.; GEORGEFF, M. P. Modeling rational agents within a bdi-architecture. **KR**, [S.l.], v. 91, p. 473–484, 1991.

REFORMAT, M.; XINWEI, C.; MILLER, J. Experiments in automatic programming for general purposes. In: TOOLS WITH ARTIFICIAL INTELLIGENCE, 2003. PROCEEDINGS. 15TH IEEE INTERNATIONAL CONFERENCE ON, 2003. **Anais...** [S.l.: s.n.], 2003. p. 366–373.

RICH, C.; WATERS, R. C. The programmer's apprentice: a research overview. **Computer**, [S.l.], v. 21, n. 11, p. 10–25, 1988.

RICH, C.; WATERS, R. C. Approaches to automatic programming. **Advances in computers**, [S.l.], v. 37, p. 1–57, 1993.

ROLIM, R. et al. Learning syntactic program transformations from examples. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 39., 2017. **Proceedings...** [S.l.: s.n.], 2017. p. 404–415.

ROZENBERG, G. (Ed.). **Handbook of graph grammars and computing by graph transformation**: volume i. foundations. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997.

RUSSELL, S. J. et al. **Artificial intelligence**: a modern approach. [S.l.]: Prentice hall Upper Saddle River, 2013. v. 3, n. 9.

SANTORINI, B. Part-of-speech tagging guidelines for the penn treebank project (3rd revision). **Technical Reports (CIS)**, [S.l.], p. 570, 1990.

SEARLE, J. R. **Expressão e significado**: estudos da teoria dos atos de fala.(tradução de ana cecília ga de camargo e ana luiza marcondes garcia). [S.l.]: São Paulo: Martins Fontes, 2002.

SEEBREGTS, C. J. et al. The openmrs implementers network. **International journal of medical informatics**, [S.l.], v. 78, n. 11, p. 711–720, 2009.

SENDALL, S.; SENDALL, S.; KÜSTER, J. Taming Model Round-Trip Engineering. **IN PROCEEDINGS OF WORKSHOP 'BEST PRACTICES FOR MODEL-DRIVEN SOFTWARE DEVELOPMENT**, [S.l.], 2004.

SHEHORY, O.; STURM, A. A brief introduction to agents. In: AGENT-ORIENTED SOFTWARE ENGINEERING, 2014. **Anais...** [S.l.: s.n.], 2014. p. 3–11.

SHIMOYAMA, M. et al. Three ontologies to define phenotype measurement data. **Frontiers in Genetics**, [S.l.], v. 3, p. 87, Apr. 2012.

SOWA, J. F. Conceptual graphs for a data base interface. **IBM Journal of Research and Development**, [S.l.], v. 20, n. 4, p. 336–357, 1976.

SOWA, J. F. et al. **Knowledge representation**: logical, philosophical, and computational foundations. [S.l.]: Brooks/Cole Pacific Grove, 2000. v. 13.

TAENTZER, G. Agg: a graph transformation environment for modeling and validation of software. In: INTERNATIONAL WORKSHOP ON APPLICATIONS OF GRAPH TRANSFORMATIONS WITH INDUSTRIAL RELEVANCE, 2003. **Anais...** [S.l.: s.n.], 2003. p. 446–453.

TAHVILDARI, L.; KONTOGIANNIS, K. A software transformation framework for quality-driven object-oriented re-engineering. In: SOFTWARE MAINTENANCE, 2002. PROCEEDINGS. INTERNATIONAL CONFERENCE ON, 2002. **Anais...** [S.l.: s.n.], 2002. p. 596–605.

TALANOV, M.; KREKHOV, A.; MAKHMUTOV, A. Automating programming via concept mining, probabilistic reasoning over semantic knowledge base of se domain. In: SOFTWARE ENGINEERING CONFERENCE (CEE-SECR), 2010 6TH CENTRAL AND EASTERN EUROPEAN, 2010. **Anais...** [S.l.: s.n.], 2010. p. 30–35.

TOUTANOVA, K.; MANNING, C. D. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In: JOINT SIGDAT CONFERENCE ON EMPIRICAL METHODS IN NATURAL LANGUAGE PROCESSING AND VERY LARGE CORPORA: HELD IN CONJUNCTION WITH THE 38TH ANNUAL MEETING OF THE ASSOCIATION FOR COMPUTATIONAL LINGUISTICS-VOLUME 13, 2000., 2000. **Proceedings...** [S.l.: s.n.], 2000. p. 63–70.

VAN HARMELEN, F.; LIFSCHITZ, V.; PORTER, B. **Handbook of knowledge representation**. [S.l.]: Elsevier, 2008. v. 1.

WAKATANI, A.; MAEDA, T. Automatic generation of programming exercises for learning programming language. In: COMPUTER AND INFORMATION SCIENCE (ICIS), 2015 IEEE/ACIS 14TH INTERNATIONAL CONFERENCE ON, 2015. **Anais...** [S.l.: s.n.], 2015. p. 461–465.

WEISS, D. M.; LAI, C. T. R. **Software product-line engineering**: a family-based software development process. [S.l.]: Addison-Wesley Reading, 1999. v. 12.

WEISS, G. Multiagent systems: a modern approach to distributed artificial intelligence. , [S.l.], 2013.

WOOLDRIDGE, M. **An introduction to multiagent systems**. [S.l.]: John Wiley & Sons, 2009.

XIAO, L. et al. Automated web service composition using genetic programming. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE WORKSHOPS (COMPSACW), 2012 IEEE 36TH ANNUAL, 2012. **Anais...** [S.l.: s.n.], 2012. p. 7–12.

XIN, Q.; REISS, S. P. Leveraging syntax-related code for automated program repair. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 32., 2017. **Proceedings...** [S.l.: s.n.], 2017. p. 660–670.