UNIVERSIDADE FEDERAL DO PAMPA

Luiz Paulo Franz

Extração Automática de Modelos Conceituais a Partir de Requisitos Para Sistemas Autoadaptativos

Alegrete
2017

Luiz Paulo Franz

Extração Automática de Modelos Conceituais a Partir de Requisitos Para Sistemas Autoadaptativos

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software.

Orientador: Prof. Mestre Joao Pablo Silva da Silva

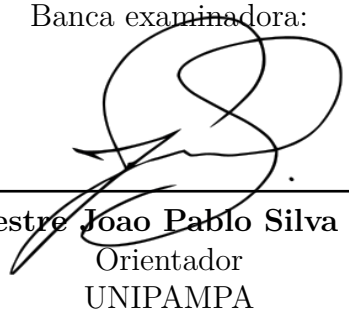
Alegrete
2017

Luiz Paulo Franz

Extração Automática de Modelos Conceituais a Partir de Requisitos Para Sistemas Autoadaptativos

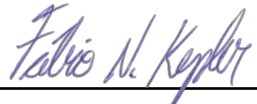
Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software.

Trabalho de Conclusão de Curso defendido e aprovado em 28 de junho de 2017.
Banca examinadora:



Prof. Mestre Joao Pablo Silva da Silva
Orientador
UNIPAMPA

Prof. Mestre Alice Fonseca Finger
UNIPAMPA



Prof. Doutor Fabio Natanael Kepler
UNIPAMPA

Humildemente dedicado a:

Armindo, a quem chamo "pai",
Elisa, a quem chamo "mãe",
Aos meus dois irmãos, Alcione e Elói, meus melhores amigos.
E acima de tudo dedico a meu outro pai, a quem chamo de "Deus".

AGRADECIMENTOS

Agradeço primeiramente a Deus, a quem tenho muito mais a agradecer do que pedir, e insisto em fazer o contrário. Obrigado por tolerar minhas falhas, e ainda me dar forças para superar meus desafios.

Agradeço aos meus familiares, sem eles não estaria aqui. Para a Anne e família, obrigado por me receberem em suas vidas e pelo conforto recebido.

Obrigado aos professores, por me ensinarem a aprender. Ao meu orientador João Pablo, por aceitar o desafio, e me desafiar a extrair o melhor de mim.

Um obrigado aos meus amigos, Luiz Felipe (Cabelo), Augusto Corrêa, Gabriel Dalenogari, Giliardi Schmidt, Marcus Norberto e Lukas Gaedicke (amigo de longa data), por tornarem a "colônia" um lugar ainda melhor de se viver.

E um obrigado aos amigos Jonas Chagas, Pedro Sebastian, Wolleson Kelm, Kaio Rezende pelo companheirismo ao longo do curso.

"Não há fatos eternos, como não há verdades absolutas."
(Nietzche, Friedrich Wilhelm)

RESUMO

A geração de modelos conceituais na etapa de Engenharia de Requisitos (ER) é uma prática que auxilia tanto no entendimento do problema quanto no projeto inicial da solução, oferecendo mecanismos para a representação visual de elementos do mundo real e seus relacionamentos. Quando o software sob desenvolvimento se trata de um Sistema Autoadaptativo (SA), temos situações de incerteza que devem ser tratadas e endereçadas na etapa de ER, tornando essa etapa mais complexa e evidenciando a carência de abordagens mais especificas. A geração dos modelos conceituais normalmente ocorre de forma manual, onde um membro da equipe modela os conceitos com base em especificações preliminares de requisitos, essa é uma atividade difícil e complexa, que consome tempo e recursos, e seus resultados são altamente dependentes da experiência do modelador. Com base nisso, uma ferramenta que automatize esse processo diminui o tempo gasto nas atividades de modelagem, minimiza erros e diminui custos. Neste trabalho desenvolvemos um protótipo de ferramenta baseada na plataforma Eclipse, com objetivo de extrair um modelo conceitual a partir de especificações textuais de requisitos para SAs expressos na linguagem RELAX. Formulamos uma série de regras de extrações que combinam técnicas de Processamento de Linguagem Natural (PLN), a semântica da linguagem RELAX e o *profile* UML RelaxML (para modelagem conceitual de requisitos RELAX) para processar os requisitos e produzir um modelo conceitual como saída. Avaliamos nossas saídas com um método baseado na área de recuperação de informações desenvolvido especificamente para ferramentas desse tipo, e nossos resultados são animadores, apresentando uma taxa de recuperação dos elementos do modelo (*recall*) de 100% e uma taxa de elementos extras (*over-specification*) em 0%, porém os dados extraídos não são muito precisos (*precision*), aproximadamente 65%. Apesar de ainda se tratar de um protótipo, consideramos que o objetivo do trabalho foi alcançado, porém os resultados indicam oportunidades de melhoria.

Palavras-chave: Modelo Conceitual. Engenharia de Requisitos. Análise de Requisitos. Processamento de Linguagem Natural. Sistemas Autoadaptativos. Automação.

ABSTRACT

The generation of conceptual models on the Requirements Engineering (ER) stage is a practice that helps on the understanding of the problem and also the initial solution design, providing mechanisms for a visual representation of real-world elements and their relationships. When the software being developed is a Self-Adaptive System (SA), we have situations of uncertainty that must be addressed in the ER stage, making this stage more complex and evidencing the lack of more specific approaches. The generation of these models usually occurs manually, where a team member models the concepts based on preliminary requirements specifications, this is a difficult and complex activity that consumes time and resources, and its results are highly dependent on the modeler's experience. Based on this, a tool that automates this process would reduce the time spent in the modeling activities, minimize errors, and reduce costs. In this work, we developed a prototype tool based on the Eclipse platform, with the objective of extracting a conceptual model from the textual specifications of requirements for SAs expressed in the RELAX language. We formulate a series of extraction rules that combine Natural Language Processing (NLP) techniques, the RELAX language semantics and the RelaxML UML profile (for RELAX conceptual requirements modeling) to process the requirements and produce a conceptual model as output. We evaluate our outputs with a method based on the area of information retrieval developed specifically for this type of tools, and our results are encouraging, we presented a model elements retrieval (recall) rate at 100% and extras elements rate (over-specification) at 0%. However, the extracted data is not very accurate (accuracy), nearly 65%. Although it is still a prototype, we consider that the objective of the work was achieved, but the results indicate opportunities for improvement.

Key-words:Conceptual Model. Requirements Engineering. Requirements Analysis. Natural Language Processing. Self Adaptive Systems. Automation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

API *Application Programming Interface*

AST *Abstract Syntax Tree*

CASE *Computer-Aided Software Engineering*

DSL *Domain Specific Language*

DSML *Domain Specific Modeling Language*

EMF *Eclipse Modeling Framework*

ER Engenharia de Requisitos

ES Engenharia de Software

FBTL *Fuzzy Branching Temporal Logic*

IDE *Integrated Development Environment*

LN Linguagem Natural

PLF Processamento de Linguagem Formal

PLN Processamento de Linguagem Natural

POS *Part-Of-Speech*

RCP *Rich CLient Platform*

SA Sistema Autoadaptativo

UML *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Softwares estão cada vez mais presentes em nosso dia a dia, os avanços na tecnologia possibilitaram que eles se tornassem mais versáteis atuando nos mais diversos ambientes. Essa versatilidade foi alcançada por meio de softwares mais autônomos, desenvolvidos para atuar sobre uma variedade de ambientes ou mesmo em situações de incerteza (JUNIOR, 2013).

Sistemas Autoadaptativos (SAs) são capazes de adequar seu estado e comportamento em resposta a mudanças no ambiente onde se encontram (JUNIOR, 2013). Whittle et al. (2010) comentam que tais sistemas devem atuar sobre um conjunto de condições e contextos diversos, o que torna difícil de prever os estados pelos quais esse software vai passar durante seu ciclo de vida. Além disso, os autores salientam que há casos em que o contexto ou estado do ambiente são imperfeitamente entendidos, e mesmo nessas condições um SA deve ser capaz de tomar decisões.

As características e incertezas inerentes aos SAs os diferenciam dos softwares convencionais, tais diferenças também ocorrem ao longo do seu processo de desenvolvimento. Soluções diferenciadas são recomendadas de modo a abordar correta e sistematicamente as incertezas. Conforme Whittle et al. (2010), algumas etapas do desenvolvimento de SAs ocorrem de maneira *ad-hoc*, indicando a carência de abordagens mais elaboradas e replicáveis para essa área.

Na mesma linha, Macías-Escrivá et al. (2013) salientam que são necessárias mais investigações direcionadas ao desenvolvimento de abordagens de Engenharia de Software (ES) orientadas aos SAs, de modo que seus resultados possam ser replicados. Lemos et al. (2011) complementam tal afirmação dizendo que atualmente pouco esforço é aplicado para estabelecer as abordagens de ES de forma adequada para SAs. A ES para sistemas tradicionais oferece pouco suporte e carece de mecanismos específicos para atuar sobre as propriedades exigidas pela autoadaptação.

Whittle et al. (2010) em seu trabalho abordam parte desse problema, com foco na Engenharia de Requisitos (ER). A ER convencional não previu fatores de incerteza ou softwares cientes de contexto, por exemplo. Com isso em mente, os autores oferecem uma linguagem para a especificação de requisitos voltada aos SAs, chamada RELAX.

De acordo com Bourque, Fairley et al. (2014), modelos conceituais auxiliam no entendimento tanto do problema a ser resolvido quanto da solução que está sendo proposta, oferecendo mecanismos para representar entidades do domínio do problema, refletindo seus relacionamentos e dependências existentes no mundo real.

O modelo conceitual normalmente consiste em um diagrama de classes *Unified Modeling Language* (UML), e é desenvolvido a partir do domínio do problema. Não aborda aspectos internos, como tipos de dados ou arquitetura, mas apenas as informações referentes ao contexto onde o sistema está inserido, descrevendo de modo visual como a equipe de análise organiza e entende as informações coletadas (WAZLAWICK, 2004).

Tais modelos UML tornam visíveis aspectos do software sob desenvolvimento, o que auxilia no entendimento e *desing* da solução, além de auxiliar na comunicação entre *stakeholders* e equipe de desenvolvimento, de modo a fazer uma ponte entre o que o *stakeholder* realmente quer com o que a equipe está entendendo, oferecendo um artefato visual para comunicação, o que torna a utilização de modelos conceituais recomendável (SHARMA; SRIVASTAVA; BISWAS, 2015).

A ER está muito mais relacionada ao entendimento do problema do que com sua solução (PFLEEGER; ATLEE, 2004). Ela apresenta uma série de atividades, como descreve Bourque, Fairley et al. (2014), sendo que as principais, de modo resumido são: Elicitação, onde ocorre a investigação e obtenção dos requisitos; Análise, onde esses requisitos são estudados e entendidos pela equipe; Especificação, onde os requisito são de alguma forma formalizados em algum documento ou modelo e por fim a Validação, onde os requisitos são validados para assegurar que a equipe entendeu os requisitos, e que estão em conformidade com os interesses dos *stakeholders*.

Uma das atividades chave recomendadas pelos autores Bourque, Fairley et al. (2014) para fase de análise de requisitos é a justamente a modelagem conceitual, o que torna essa atividade parte integrante da ER. Pressman (2011) chama esses modelos de "modelos de análise", que tem como propósito fornecer informações sobre as entidades e relacionamentos do sistema sob desenvolvimento, auxiliando tanto na comunicação quanto no entendimento do problema. Ainda segundo o autor, tais modelos são mutáveis e evoluem juntamente com entendimento dos analistas, para posteriormente tornar-se o próprio modelo da fase de projeto.

1.1 Motivação

Requisitos são geralmente expressos de maneira textual e a sua transcrição para um modelo é em muitos casos um processo manual realizado pelos analistas. Deeptimahanti e Sanyal (2011) comentam que a transição entre a especificação para os modelos envolve atividades complexas e difíceis, e que qualquer erro nesse processo pode ser problemático e caro de resolver posteriormente.

Dada a criticidade dessa tarefa, um analista deve realizar a transcrição de tais requisitos para os modelos com cuidado, consumindo tempo e recursos, tornando o processo lento e caro. Mesmo assim, essa atividade está suscetível a erros e qualquer perda de informações pode comprometer o projeto. Assim, o problema está na forma como esses modelos são criados atualmente, geralmente de modo manual, o que torna seus resultados altamente dependentes da experiência do modelador.

Com isso em mente, fica evidente a relevância e potencial valor de algum mecanismo de automação da modelagem conceitual. Uma ferramenta que partindo dos requisitos textuais como entrada seja capaz de identificar os elementos de um modelo conceitual, pode reduzir o tempo e os recursos necessários para realização dessa tarefa,

diminuindo assim o custo, além de minimizar alguns erros de modelagem.

Esse tema já é de interesse dos pesquisadores, como Elbendak, Vickers e Rossiter (2011), Sharma, Srivastava e Biswas (2015), Sagar e Abirami (2014) e Afreen, Bajwa e Bordbar (2011) por exemplo, porém nenhum deles direcionou seus estudos para os SAs, que possuem uma ER um tanto mais complexa devido a sua natureza.

1.2 Objetivos

Dado o contexto e a motivação, temos como objetivo geral desse trabalho o desenvolvimento de uma ferramenta que realiza a extração de modelos conceituais a partir de requisitos para SAs escritos na linguagem RELAX. Abaixo listamos os objetivos específicos:

- explorar o formalismo da linguagem RELAX;
- desenvolver um conjunto de regras para a extração de modelos conceituais dos requisitos;
- integrar nosso extrator à ferramenta *Relax Editor* originalmente proposta por Moro (2015);

1.3 Organização do Documento

O restante do trabalho é estruturado conforme segue:

- Capítulo 2: Fundamentação Teórica, onde os conhecimentos necessários para o entendimento do estudo são apresentados;
- Capítulo 3: Base Tecnológica, apresentamos as ferramentas e tecnologias utilizadas em nossa abordagem;
- Capítulo 4: Trabalhos Relacionados, apresenta a estratégia de pesquisa utilizada em nossa revisão bibliográfica, bem como seus resultados, exibindo uma série de estudos anteriores que serviram de base para o presente trabalho;
- Capítulo 5: Extração dos Conceitos, onde descrevemos como utilizamos as teorias e tecnologias discutidas nos capítulos anteriores para desenvolver um protótipo funcional de um extrator de modelos conceituais para SAs;
- Capítulo 6: Validação do Protótipo, apresentamos e explicamos uma forma de avaliação para nosso protótipo, bem como seus resultados e discussões;
- Capítulo 7: Conclusões, onde pontuamos os principais desafios enfrentados no decorrer do trabalho, as lições aprendidas e uma síntese do resultado, e por fim indicamos possíveis trabalhos futuros;

2 FUNDAMENTAÇÃO TEÓRICA

Nesse capítulo apresentamos os fundamentos e teorias que servem como base para a compreensão deste trabalho. A seção 2.1 discute sobre os SAs e suas propriedades. Na seção 2.2 o tópico "Modelos Conceituais" presente na ER é discutido mais a fundo. Na seção 2.3 são apresentadas de forma sucinta as etapas do Processamento de Linguagem Formal (PLF) que ocorrem nos compiladores. Na seção 2.4 abordamos os temas e conceitos necessários para o entendimento e utilização de tecnologias e técnicas de Processamento de Linguagem Natural (PLN) e por fim na seção 2.5 apresentamos algumas lições aprendidas e pontuamos os principais tópicos discutidos no capítulo.

2.1 Sistemas Autoadaptativos

Softwares estão cada vez mais presentes em nosso dia a dia, e vêm crescendo em tamanho e complexidade. Os avanços na tecnologia possibilitaram que os softwares se tornassem mais versáteis atuando nos mais diversos ambientes. Com base nisso, são necessários softwares mais autônomos, desenvolvidos de modo a agir sobre uma variedade de ambientes, mesmo em situações de incerteza, os SAs buscam suprir essa necessidade (JUNIOR, 2013).

Tais sistemas devem lidar com ambientes em constante mudança, o que pode gerar requisitos que não foram previstos em seu projeto (BRUN et al., 2009). Complementando, Macías-Escrivá et al. (2013) os definem como softwares capazes de avaliar e mudar seu próprio comportamento, tomando decisões em tempo de execução com base em seu ambiente e seu estado atual. Por fim Brun et al. (2009) afirmam que tais softwares são capazes de ajustar seu comportamento em resposta ao ambiente de forma automática.

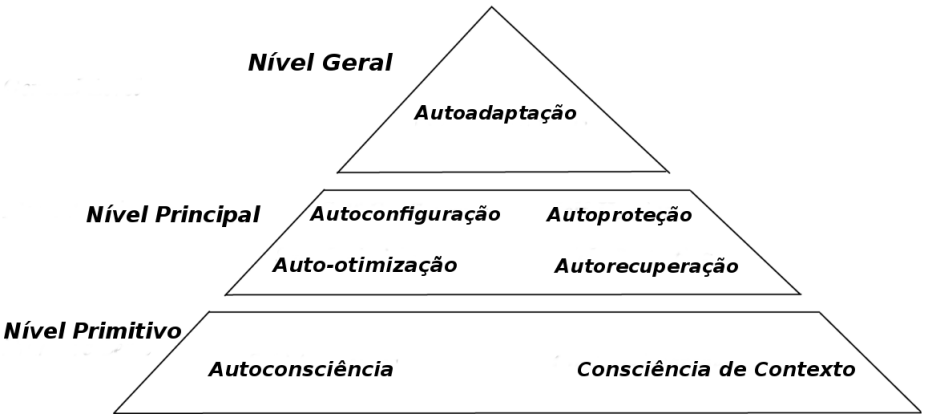
Whittle et al. (2010) comentam que tais sistemas devem atuar sobre um conjunto de condições e contextos diversos o que torna difícil de prever os estados pelo qual esse software vai passar durante seu ciclo de vida. Além disso os autores salientam que há casos em que o contexto e/ou estado do ambiente são imperfeitamente entendidos pelos analistas, e mesmo nessas condições um SA deve ser capaz de tomar decisões.

As características intrínsecas dos SAs fazem com que as práticas tradicionais de ES sejam insuficientes ou impróprias para o desenvolvimento desses sistemas. Portanto, a ES deve encontrar meios inovadores para lidar com essas características, de modo a abordar esses pontos de maneira sistemática.

Para que um software seja considerado um SA, o mesmo deve apresentar uma ou mais propriedades de autoadaptação. Conforme Salehie e Tahvildari (2009), essas propriedades são conhecidas como "auto-*", e são elas: Autoconfiguração; Autorecuperação; Auto-otimização e Autoproteção. Sendo que a propriedade transversal a todas essas é a Autoadaptação. Essas propriedades são derivadas de atributos externos, como o ambiente (*context-awareness*) e atributos internos, como o estado do sistema (*self-awareness*), a Figura 1 exibe uma estrutura hierárquica contendo tais propriedades, vejamos mais

detalhes das propriedades apresentadas por Salehie e Tahvildari (2009).

Figura 1 – Classificação hierárquica das propriedades da autoadaptação



Fonte: (SALEHIE; TAHVILDARI, 2009)

O **Nível Primitivo (*Primitive Level*)** apresenta as propriedades básicas que sustentam a autoadaptação e está dividido em duas:

1. Autoconsciência (*self-awareness*): Significa que o software é consciente de si próprio, conhecendo suas ações e estado.

2. Consciência de Contexto (*context-awareness*): Significa que o software é ciente do contexto onde está inserido, ou seja, do seu ambiente.

No **Nível Principal (*Major Level*)** as propriedades de autoadaptação devem ser executadas, sendo divididas em quatro propriedades:

1. Autoconfiguração (*self-configuring*): é a capacidade de o software se configurar automaticamente em respostas ao ambiente, integrando, atualizando ou manipulando entidades do software;

2. Autorecuperação (*self-healing*): é a capacidade de descobrir, diagnosticar e reagir à disrupções, de modo a evitar falhas e problemas maiores;

3. Auto-otimização (*self-optimizing*): também conhecido como autoajuste, representa a capacidade de alterar a performance e a utilização dos recursos de forma a atingir determinados objetivos (requisitos), como por exemplo, entrar em modo de espera para economizar energia;

4. Autoproteção (*self-protecting*): é a capacidade de detectar brechas de segurança e se recuperar de seus efeitos;

O **Nível Geral (*General Level*)** contém a visão global do software e possui uma única propriedade transversal, a **Autoadaptação**, a qual depende das propriedades abaixo dela na hierarquia.

2.2 Modelos Conceituais

A modelagem conceitual é uma atividade integrante e recomendada pela ER, sendo o modelo conceitual um dos artefatos previstos para a etapa de análise de requisitos (BOURQUE; FAIRLEY et al., 2014). Consiste na elaboração de modelos para auxiliar no processo de entendimento do domínio do software, tanto para a equipe de desenvolvimento quanto para os *stakeholders*. Além disso tais modelos auxiliam no processo de comunicação entre as partes, possibilitando um comum entendimento por meio desses artefatos.

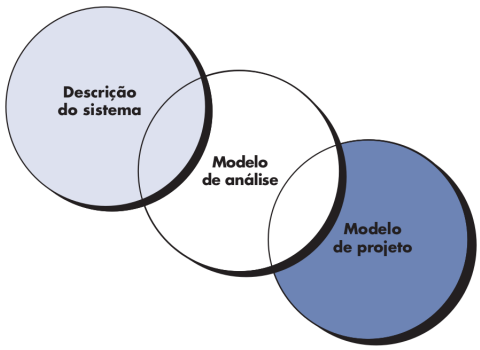
Pressman (2011) cita que a análise de requisitos é importante pois nessa fase o *stakeholder* pode ainda não estar certo do que é realmente necessário em certos pontos do sistema. Do mesmo modo o desenvolvedor pode estar incerto sobre quais abordagens usar, impossibilitando uma especificação completa. Nesse ponto, os modelos apresentam um papel fundamental, melhorando a comunicação e o entendimento do problema. Seu foco deve estar no "o que" e não no "como".

Bourque, Fairley et al. (2014) apontam a modelagem conceitual como uma das atividades chave para uma boa análise de requisitos, auxiliando no entendimento da situação onde o problema ocorre e da solução sob desenvolvimento. Tais modelos representam as entidades do domínio do problema, organizados de uma forma que essas reflitam seus atributos e relacionamentos existentes no mundo real. Os autores comentam que os modelos mais usados nessa etapa são os baseados na UML, e recomendam a criação de um modelo inicial do contexto onde o software está inserido, de modo que as conexões entre o software e seu ambiente sejam expostas.

Os modelos criados na fase de requisitos são as primeiras representações técnicas de um sistema. "A palavra escrita é um maravilhoso veículo para a comunicação, porém, não é necessariamente a melhor maneira de representar os requisitos de um software" (PRESSMAN, 2011). Além disso, o autor comenta que ocorre um pouco de projeto dentro da fase de ER e ocorre um pouco de ER dentro da fase projeto, como fica claro na Figura 2. Os modelos desenvolvidos nessa etapa podem servir como uma ponte para a posterior fase de projeto,

Um modelo conceitual normalmente consiste em um diagrama de classes UML desenvolvido a partir do domínio do problema. Não aborda aspectos internos do sistema, como tipos de dados ou arquitetura, mas apenas as informações referentes ao contexto onde o sistema está inserido, isso descreve de modo visual como a equipe de análise organiza e entende as informações coletadas (WAZLAWICK, 2004). Completando, Pressman (2011) indica que tais modelos são mutáveis e evoluem juntamente com entendimento dos analistas, para posteriormente tornar-se o próprio modelo da fase de projeto.

Figura 2 – Sobreposição das fases de ER e projeto.



Fonte: (PRESSMAN, 2011)

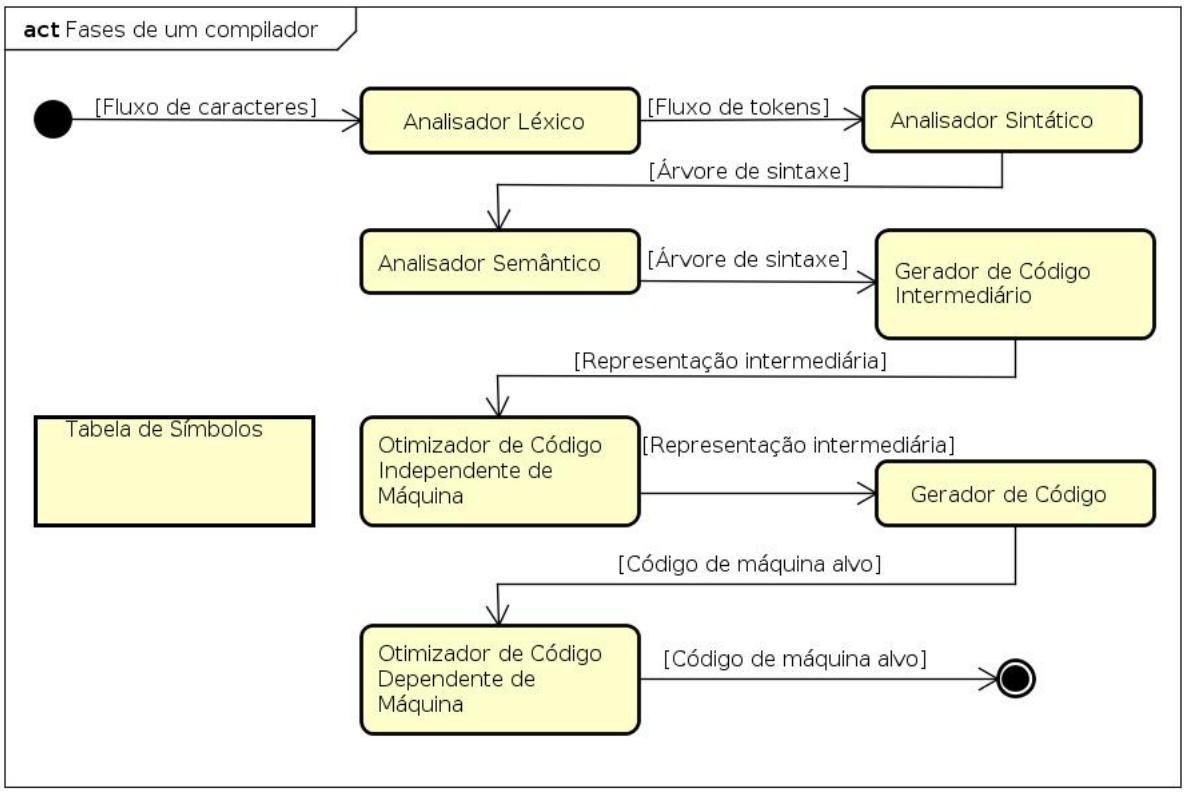
2.3 Processamento de Linguagens Formais

O PLF é normalmente executado por um software conhecido como compilador, que atua sobre uma linguagem pré-definida, ou seja, esse tipo de processador já conhece previamente toda a estrutura da linguagem que será processada. Portanto diferentemente do que ocorre com o PLN (ver seção 2.4), nesse caso a gramática da linguagem já é conhecida, na verdade a linguagem só pode ser considerada "formal" se possuir uma gramática correspondente capaz de representá-la em sua totalidade (BENDER, 1996).

Gramática, segundo Bender (1996), é um conjunto de regras que definem a sintaxe de uma linguagem. As linguagens formais são definidas por uma gramática oficial e definitiva, e é ela quem define as regras que uma sentença deve respeitar para ser considerada parte da linguagem. De acordo com Russell e Norvig (2004), linguagens formais são definidas como um conjunto de palavras possivelmente infinito, e possuem definições matemáticas rígidas, conhecidas como gramáticas. Em outras palavras, a gramática é quem define a ordem dos lexemas da linguagem, indicando por exemplo que o tipo de uma variável vem antes do seu identificador. Gramática é um conceito essencial na área de processamento de linguagens, tanto formais quanto naturais.

A estrutura fundamental de um compilador é apresentada na Figura 3, e conforme ela, na primeira fase, **análise léxica**, o compilador mapeia o fluxo de caracteres em sequências significativas, chamadas lexemas, normalmente utilizando de expressões regulares. Para cada lexema encontrado, o analisador léxico gera um *token* como saída, que por sua vez é um elemento atômico da linguagem, que pode ser uma palavra-chave (*class*), ou um identificador válido (como o nome de uma classe) por exemplo. Cada *token* segue o seguinte modelo: <nome-*token*, valor-atributo>, um exemplo de *token* é <id, 2>, onde o nome-*token* é um símbolo abstrato utilizado para a posterior análise sintática, e o valor-atributo aponta para uma entrada em uma tabela de símbolos. A tabela de símbolos é necessária para posterior análise semântica e geração de código (AHO; ULLMAN;

Figura 3 – Fases de um compilador.



Fonte: (AHO; ULLMAN; SETHI, 2008)

SETHI, 2008).

A segunda fase, **análise sintática** (também chamada de *parsing*), utiliza apenas os nome-*tokens* produzidos na análise léxica para criar uma estrutura de árvore chamada *Abstract Syntax Tree* (AST), que possibilita verificar se a ordem dos *tokens* está correta, por exemplo, é a análise sintática que certifica que o tipo de uma variável deve aparecer antes de seu identificador (AHO; ULLMAN; SETHI, 2008).

Um código nem sempre pode ser considerado correto apenas com a análise sintática. Uma das situações que não podem ser verificadas nassa etapa é o *type checking*, ou seja, verificações dos tipos para certificar que uma variável do tipo *Integer* não receba uma *String* como entrada por exemplo, outra situação diz respeito ao escopo das variáveis (BETTINI, 2016). Essas e outras análises ocorrem na terceira fase.

A terceira fase é a **análise semântica**, que utiliza a AST produzida na etapa anterior e a tabela de símbolos produzida na análise léxica para verificar a consistência das expressões com o sentido da linguagem, fazendo análises nos nós da árvore, onde cada nó é uma pequena parte da linguagem. Se a árvore está correta, tanto sintática quanto semanticamente, ela pode ser usada para a geração de código (AHO; ULLMAN; SETHI,

2008), (BETTINI, 2016).

Na geração de **código intermediário**, quarta fase, o compilador pode produzir uma ou mais representações intermediárias dependendo de sua implementação. A geração desse código intermediário ocorre a partir da árvore sintática (AST) e deve ser facilmente produzido e traduzido para a máquina alvo. Normalmente o código intermediário segue o padrão *assembler* de instruções, que é produzido para facilitar a posterior geração do código objeto. É nesse código intermediário que a otimização das instruções é realizada, essa otimização ocorre de acordo com a arquitetura na qual o compilador está sendo executado e de acordo com o objetivo da otimização do código, como por exemplo o ganho de velocidade, ou reduzir o consumo de energia. A otimização caracteriza a quinta etapa do processo de compilação (AHO; ULLMAN; SETHI, 2008).

Por fim, temos a sexta fase, **geração de código para a máquina alvo**, que traduz o código intermediário para o código de máquina, nesse código de máquina há uma última fase de **otimização**.

2.4 Processamento de Linguagem Natural

Jurafsky e Martin (2009) definem PLN como abordagens computacionais que tem por objetivo possibilitar que um computador raciocine sobre a linguagem falada e escrita por nós humanos, conhecida como Linguagem Natural (LN). De modo a realizar tarefas úteis como a comunicação entre máquinas e humanos ou realizar alguma ação sobre algum texto.

Como dito na seção 2.3, as linguagens formais são definidas por uma gramática oficial e definitiva, porém isso não ocorre no caso das LNs, essas não apresentam uma definição rígida. Para possibilitar seu processamento computacional, as LNs devem ser tratadas com alguma rigidez, ou seja, princípios formais devem ser "inseridos" à LN. Jurafsky e Martin (2009) afirmam que **não existe uma gramática capaz de oferecer formalismo que corresponda a uma LN em sua totalidade**, e mesmo quando limitada, a natureza livre das LNs dificultam um mapeamento formal e completo. Porém os autores salientam que é possível extrair ou modelar uma gramática para a LN quando a mesma é aplicada em um ambiente controlado e restrito.

O termo *parsing* de acordo com Jurafsky e Martin (2009) é amplamente utilizado na área de processamento de linguagens. Em PLN, de modo geral, isso significa receber algum *input* em LN como entrada e gerar algum tipo de estrutura linguística com essa informação como saída. Portanto, em PLN existem vários tipos de *parsing*, listaremos dois deles.

O ***constituency parsing***, tem por objetivo mapear e obter as estruturas gramaticais das sentenças para uma representação sintática, sendo o tipo de *parsing* mais comumente usado. Outro *parsing* interessante é o ***dependency parsing***, que busca mapear e obter os relacionamentos de dependência entre as palavras, nesse caso os re-

lacionamentos são mais importante que suas estruturas gramaticais. Esses dois tipos de *parsing* apresentam algumas etapas em comum, vejamos:

2.4.1 Tokenization

Pode também ser chamado de segmentador de palavras (*word segmentation*). Em linhas gerais, o objetivo desse processo é separar o texto em palavras ou sentenças menores chamadas *tokens*. Esse processo leva em conta espaços em branco para delimitar palavras e regras de pontuação para segmentação de sentenças, porém isso unicamente não é suficiente (JURAFSKY; MARTIN, 2009).

Alguns *tokens* apresentam espaços em seu interior, como por exemplo o substantivo próprio *The New York Times* (jornal norte americano). Isso deve ser considerado como um único *token*. Siglas como Ph.D. seguem a mesma lógica, apresentando pontuação em seu interior, portanto essas particularidades devem ser tratadas (JURAFSKY; MARTIN, 2009).

Em outras palavras, o principal objetivo dessa etapa é encontrar os lexemas de uma LN, quebrando a entrada em **sentenças** e em **palavras**.

2.4.2 Part-Of-Speech Tagging

Os *Part-Of-Speech* (POS) são elementos de linguagem com papéis distintos que quando combinados possibilitam a comunicação. Cada palavra de um texto em LN possui um POS, alguns exemplos são substantivos, verbos, preposições, advérbios, entre outros. Podem também ser chamados de classe gramatical ou de *tag*.

Jurafsky e Martin (2009) atestam que a importância dos POS para o PLN está no montante de informações que eles fornecem sobre as palavras (*tokens*) e seus vizinhos. Cada linguagem tem suas características próprias, assim esse conjunto de POS é dependente da linguagem, apesar de algumas *tags* serem comuns a várias LNs (substantivos e verbos ocorrem tanto no inglês quanto no português por exemplo). A esse conjunto de POS damos o nome de *tagset*, a Figura 4 apresenta um *tagset* comumente usado na língua inglesa.

O processo de POS *tagging*, ou apenas *tagging*, está relacionado a esses conceitos e consiste em assinar cada *token* de um texto com sua *tag* correspondente, recebendo como *input* uma *string* de palavras e um *tagset*, e o *output* gerado é essa *string* com uma única *tag* associada a cada *token*. Porém, essa nem sempre é uma tarefa simples, a LN é muitas vezes ambígua, como por exemplo a palavra “porco”, pode se referir ao animal, atuando como um substantivo, ou pode se referir a alguém que não tem boa higiene pessoal, atuando como adjetivo. Portanto, regras de desambiguação devem atuar nesses casos, analisando o contexto atual do *token*, verificando seus vizinhos e determinando sua posição e papel na sentença (JURAFSKY; MARTIN, 2009).

Figura 4 – *Tagset* do *Penn Treebank*, incluindo pontuação.

Tag	Description	Example	Tag	Description	Example
CC	coordin. conjunction	*and, but, or*	SYM	symbol	+,%, &
CD	cardinal number	*one, two*	TO	"to"	*to*
DT	determiner	*a, the*	UH	interjection	*ah, oops*
EX	existential 'there'	*there*	VB	verb base form	*eat*
FW	foreign word	*mea culpa*	VBD	verb past tense	*ate*
IN	preposition/sub-conj	*of, in, by*	VBG	verb gerund	*eating*
JJ	adjective	*yellow*	VBN	verb past participle	*eaten*
JJR	adj., comparative	*bigger*	VBP	verb non-3sg pres	*eat*
JJS	adj., superlative	*wildest*	VBZ	verb 3sg pres	*eats*
LS	list item marker	*1, 2, One*	WDT	wh-determiner	*which, that*
MD	modal	*can, should*	WP	wh-pronoun	*what, who*
NN	noun, sing. or mass	*llama*	WP$	possessive wh-	*whose*
NNS	noun, plural	*llamas*	WRB	wh-adverb	*how, where*
NNP	proper noun, sing.	*IBM*	$	dollar sign	*$*
NNPS	proper noun, plural	*Carolinas*	#	pound sign	*#*
PDT	predeterminer	*all, both*	"	left quote	' or "
POS	possessive ending	*'s*	"	right quote	' or "
PRP	personal pronoun	*I, you, he*	(	left parenthesis	[, (, {, <
PRP$	possessive pronoun	*your, one's*	)	right parenthesis	],), }, >
RB	adverb	*quickly, never*	,	comma	,
RBR	adverb, comparative	*faster*	.	sentence-final punc	. ! ?
RBS	adverb, superlative	*fastest*	:	mid-sentence punc	: ; ... – -
RP	particle	*up, off*			

Fonte: (JURAFSKY; MARTIN, 2009)

A seguir apresentamos um exemplo de uma sentença na língua inglesa apresentada por Jurafsky e Martin (2009), onde cada *token* apresentado é assinado com sua respectiva *tag*. Ilustramos o *token* assinado com uma barra seguido de sua *tag* escrita em maiúsculo:

The/DT grand/JJ /jury/NN commented/VBD on/IN a/DT number/NN of/IN other/JJ topics/NNS./.

Esse é um exemplo de saída de um processo de POS *tagging*, e serve como entrada para outras etapas subsequentes. O *tagset* usado para esse exemplo é mostrado na Figura 4.

2.4.3 Constituency Parsing

POS *tagging* oferece informações importantes sobre as palavras, porém o foco do processo de *tagging* está na análise individual dos *tokens*, mas sabemos que no processo de comunicação, as palavras não atuam individualmente, e sim de modo coletivo.

De acordo com Jurafsky e Martin (2009), um conceito importante em PLN são

os *constituents*, sua ideia fundamental é que grupos de *tokens* possuem comportamento único. Assim uma sentença é formada por um conjunto de *constituents*, que por sua vez são formados por palavras ou *tokens*, construindo de forma hierárquica uma sentença, podemos dizer que os *constituent* são as camadas intermediárias dessa estrutura.

Um exemplo simples de *constituent* são as frases substantivo, *Noun Phrase* ou NP em inglês, como "billy o cavalo" e "uma casa", perceba que mesmo possuindo mais de uma palavra, a frase toda se comporta como substantivo.

De acordo com Russell e Norvig (2004), o *constituency parsing* consiste em construir o que os autores chamam de árvore de análise ou *syntatic parse tree* a partir de uma sentença, onde os nós interiores são chamados de *constituents* e são elementos semânticos naturais que auxiliam na composição e análise de uma sentença válida, assim podemos acrescentar camadas semânticas até chegarmos às folhas da árvore, onde estão os *tokens*. Com base nessa árvore, é possível realizar a interpretação semântica de forma computacional.

Ainda segundo os autores, a construção do *syntatic parse tree* de uma sentença usa regras para combinar os *tokens* em *constituents*, e combinar *constituents* entre si, montando uma estrutura hierárquica até atingir um único nó (raiz), que deriva todos os outros. O resultado é uma estrutura de árvore que representa a sintaxe de uma sentença, que é equivalente a sua gramática livre de contexto.

O principal objetivo desse tipo de *parsing* é gerar uma estrutura gramatical pura, representada em uma estrutura de árvore chamada *syntatic parse tree* (análogo ao AST apresentado na seção 2.3). Essa estrutura é útil em tarefas como tradução computacional automática entre diferentes LNs, ou mesmo a compreensão computacional de LN, em tarefas como essa, a estrutura sintática é mais importante o relacionamento entre as palavras (SCHUSTER; MANNING, 2016).

2.4.4 Dependency Parsing

Esse tipo de *parsing* é relativamente mais simples que o anterior. Ele também utiliza as informações obtidas por meio dos processos de *Tokenization* e POS *Tagging*, porém nesse caso, *constituents* e as estruturas das frases não desempenham nenhum papel importante. Ao invés disso, a sentença tem sua estrutura sintática descrita em termos das palavras e uma semântica binária, que indica uma relação sintática entre duas palavras (JURAFSKY; MARTIN, 2009).

Um *parsing* por dependências analisa como um *token* se relaciona com outro na sentença, esse relacionamento é chamado binário pois os relacionamentos são mapeados sempre entre duas palavras, uma é a *head*, que possui papel dominante na relação, e outra palavra que é *dependent*, como o nome sugere, depende de *head* (BIRD; KLEIN; LOPER, 2009). Esse tipo de análise foca, por exemplo, no relacionamento entre um sujeito e o predicado de uma sentença, que está baseado em relações léxicas e não em relações de

constituência.

Diferentemente do que ocorre no *parsing* de constituência, a estrutura gerada nesse caso é um grafo direcionado linear, ou seja, é uma estrutura bem mais simples de se manipular visto que é linear. Os nós do grafo são as palavras, e o arcos representam as relações de dependências entre um nó *head* e outro nó *dependent*, os arcos apresentam um *label*, informando qual é a dependência entre essas palavras, a Figura 5 apresenta a lista de dependências conhecido como *Universal Dependencies*.

O *head* de uma sentença normalmente é um verbo, e todas as outras palavras dessa sentença dependem desse nó *head* ou se conectam a ele por meio de um caminho de dependências (um ou mais arcos). (JURAFSKY; MARTIN, 2009), (BIRD; KLEIN; LOPER, 2009).

Figura 5 – Dependências do *Universal Stanford Dependencies*

Core dependents of clausal predicates		
Nominal dep	*Predicate dep*	
nsubj	csubj	
nsubjpass	csubjpass	
dobj	ccomp	xcomp
iobj		
Non-core dependents of clausal predicates		
Nominal dep	*Predicate dep*	*Modifier word*
	advcl	advmod
	nfincl	neg
nmod	ncmod	
Special clausal dependents		
Nominal dep	*Auxiliary*	*Other*
vocative	aux	mark
discourse	auxpass	punct
expl	cop	
Coordination		
conj	cc	

Noun dependents		
Nominal dep	*Predicate dep*	*Modifier word*
nummod	relcl	amod
appos	nfincl	det
nmod	ncmod	neg
Compounding and unanalyzed		
compound	mwe	goeswith
name	foreign	
Case-marking, prepositions, possessive		
case		
Loose joining relations		
list	parataxis	remnant
dislocated		reparandum
Other		
Sentence head	*Unspecified dependency*	
root	dep	

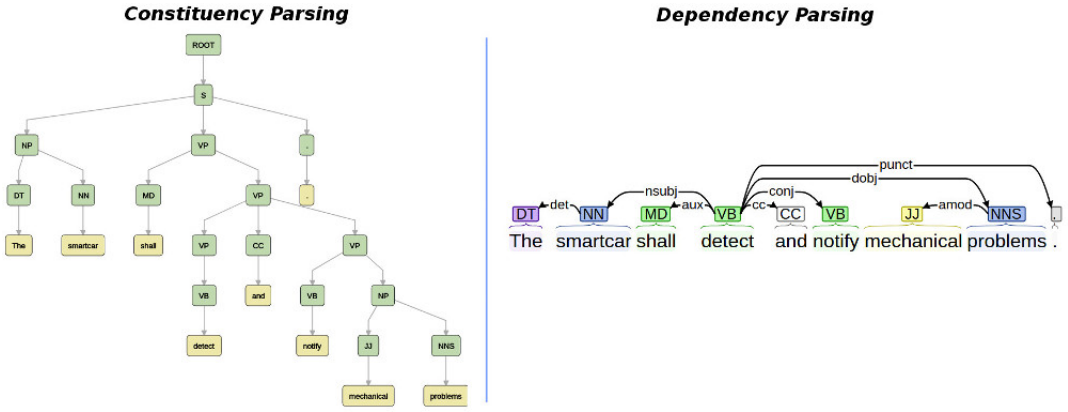
Fonte: (MARNEFFE et al., 2014)

Um grafo de dependências é projectivo se uma palavra, e todas as suas descendentes formam uma sequência contínua de palavras, ou seja, os arcos que indicam as dependências não podem se cruzar, essa abordagem simplifica o algoritmo e a estrutura gerada, e continua apresentando estruturas úteis para várias tarefas de PLN, como a mineração e extração de dados por exemplo (BIRD; KLEIN; LOPER, 2009).

O principal objetivo desse tipo de *parsing*, segundo Schuster e Manning (2016), é encontrar os relacionamentos entre as palavras de uma sentença, o que pode ser mais útil, ou pelo menos suficiente, para atividades de mineração e extração de dados.

Por fim podemos verificar a diferença entre as saídas para a entrada "*The smartcar shall detect and notify mechanical problems.*" de um *constituency parser* e um *dependency parsing* na Figura 6, observe que a saída do *dependency parsing* não apresenta nenhum nó não-terminal, tornando sua estrutura mais simples, mas ainda poderosa.

Figura 6 – Diferenças entre o *constituency parsing* e o *dependency parsing*.



Fonte: O autor

2.5 Lições do Capítulo

Os fundamentos que exploramos nesse capítulo permitem um entendimento sobre os temas utilizados para a realização desse trabalho. Começamos pelos SAs, que apresentam uma ampla variedade de aplicações, e são sistemas inteligentes, capazes de mudar seu comportamento com base em informações de seu ambiente.

Porém o processo de desenvolvimento de SAs é diferenciado, devido suas características, nesse sentido evidenciamos que as abordagens da ER tradicional quando aplicadas a SAs são insuficientes.

Evidenciamos também que a modelagem conceitual é parte integrante da ER, e que esta atividade é recomendada por uma série de autores. Pontuamos que a tarefa de criar tais modelos é crítica, o que consome recursos, e mecanismos para auxiliar nesse sentido são recomendados. O processo de criação de modelos conceituais normalmente usa como base especificações preliminares de requisitos, que podem estar descritos em linguagens formais ou em LN. O PLF e PLN abordam justamente o processamento computacional de linguagens, e apresentamos a base de como cada um ocorre.

3 BASE TECNOLÓGICA

Neste capítulo apresentamos as tecnologias utilizadas no desenvolvimento deste trabalho. A seção 3.1 apresenta uma visão geral da Plataforma Eclipse, sua arquitetura modular que permite estender suas funcionalidades. A seção 3.2 esta dividida em duas partes, a primeira apresenta a ferramenta Xtext, para o processamento de linguagens formais, a segunda seção apresenta a ferramenta *Stanford Parser* que é amplamente utilizada para o PLN. Na seção 3.3 apresentamos a linguagem RELAX, que se apresenta como uma boa opção para a ER no contexto de SAs e por fim a seção 3.4 apresenta as lições aprendidas e considerações sobre este capítulo.

3.1 Plataforma Eclipse

A plataforma Eclipse tem como veículo chefe uma IDE amplamente utilizado na indústria e academia, é um projeto *open-source* e vai além do IDE eclipse o qual estamos mais habituados, seu início remota ao ano de 2001 (VOGEL; MILINKOVICH, 2013). Foi criado com o propósito de oferecer uma plataforma comum para os diversos tipos de IDE, apresenta uma arquitetura que possibilita sua extensão para necessidades específicas, através da integração de módulos externos à plataforma chamados *plugins*. Para isso é disponibilizado uma *Application Programming Interface* (API) bem definida que possibilita a integração de novas funcionalidades ao corpo da plataforma. De acordo com Rivières e Wiegand (2004) é "uma IDE para qualquer coisa e para nada em particular".

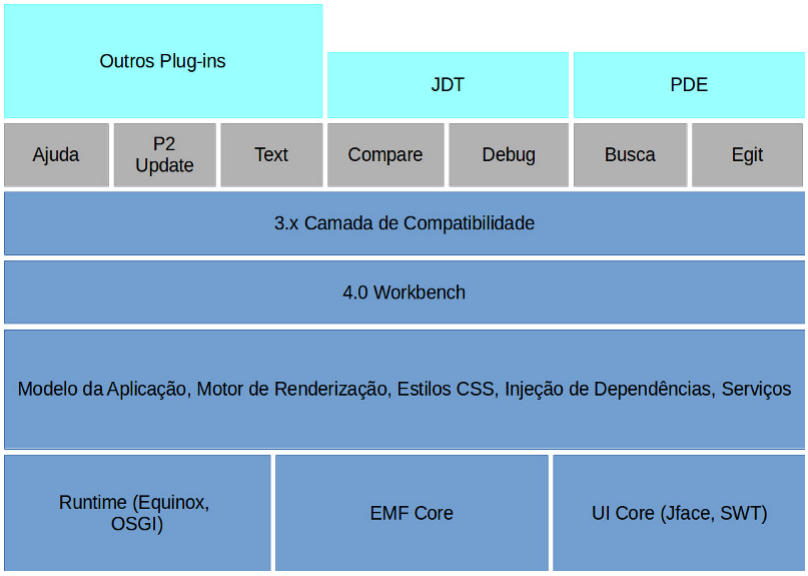
No ano de 2004 a versão 3.0 da plataforma Eclipse foi lançada, a qual trouxe uma novidade, o desenvolvimento de aplicações *stand-alone* baseadas na plataforma Eclipse. Isso possibilitou a criação de produtos *eclipse-like* customizados, que utilizam como base os recursos providos pela plataforma, a essas aplicações *stand-alone* foi dado o nome de Eclipse RCP. Atualmente a plataforma Eclipse está em sua versão 4, que trouxe várias mudanças em relação a versão 3, como injeção de dependências e suporte a CSS por exemplo (VOGEL; MILINKOVICH, 2013).

A plataforma Eclipse foi desenvolvida seguindo uma arquitetura altamente modular, a Figura 7 apresenta a arquitetura do Eclipse IDE, que contém vários recursos dos quais explicaremos os principais.

Runtime: É acionado na inicialização do IDE e sua principal função é identificar os *plugins* registrados na ferramenta. Como aplicações eclipse são compostas por uma série de *plugins*, essa é uma parte importante da aplicação. A arquitetura em *plugins* é altamente modular, e *Open Services Gateway Initiative* (**OSGI**) é uma especificação de uma abordagem modular para Java, a qual o **Equinox** implementa na plataforma Eclipse (VOGEL; MILINKOVICH, 2013).

UI Core: Responsável por apresentar os componentes visuais, utilizando a biblioteca de componentes visuais em Java *Standart Widget Toolkit* (SWT), que é acessada através de uma API padrão, conhecida como **Jface**.

Figura 7 – Arquitetura geral do Eclipse IDE



Fonte: (VOGEL; MILINKOVICH, 2013)

4.0 *Workbench*: É o ambiente principal da ferramenta, representa uma abstração da aplicação em termos visuais, aqui é representado o modelo de aplicação (*Application Model*), com suporte a CSS, injeção de dependências e é onde o motor de renderização se encontra (VOGEL; MILINKOVICH, 2013).

3.x Camada de compatibilidade: Camada de compatibilidade para *plugins* desenvolvidos seguindo a antiga arquitetura.

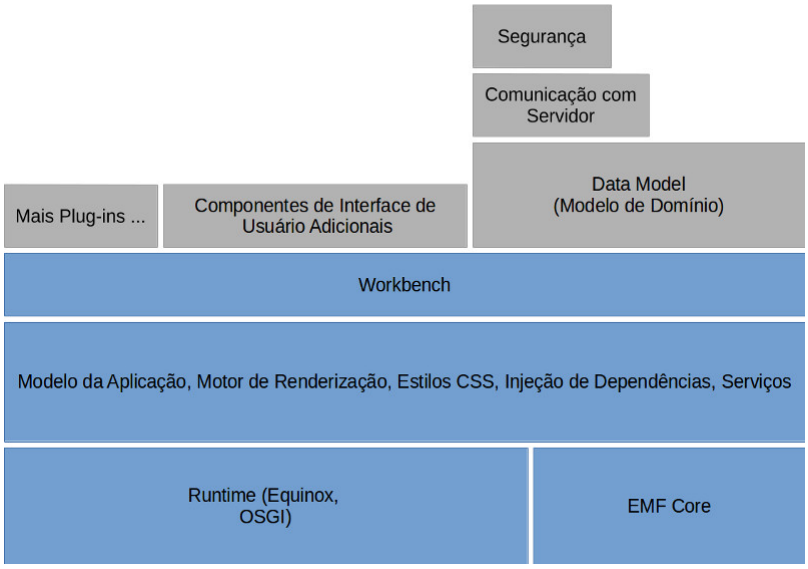
EMF Core: O *Eclipse Modeling Framework* (EMF) é um conjunto de *plugins* com ferramentas para representação de modelos e respectiva geração de código, que possibilita o desenvolvimento de um modelo de domínio, seu objetivo é possibilitar o desenvolvimento baseado em modelos. A partir desses modelos o EMF fornece ferramentas e suporte *runtime* para produção de classes Java que os implementam. Além disso, o EMF pode ser utilizado para o desenvolvimento de ferramentas de modelagem. Em outras palavras, esse é um recurso que possibilita a edição e utilização de um modelo em tempo de execução (FOUNDATION, 2017a).

Outros Plugins: Onde se encontram e se conectam os *plugins* externos, desenvolvidos por terceiros.

A arquitetura de um Eclipse *Application*, ou aplicação RCP é apresentada da Figura 8, ela usa parte da arquitetura exibida na Figura 7, mantendo os elementos base necessários para um produto eclipse, como o *Workbench* e a plataforma *Runtime*. Porém tira uma série de elementos que são opcionais, adicionando outros, como o Modelo de Dados, que é processado pelo EMF *core*, além de recursos para garantir a segurança da plataforma *Runtime* e a possibilidade de integração com serviços online. Em outros

plugins, podemos adicionar nossos próprios *plugins*, ou de terceiros. Por fim, uma aplicação RCP possibilita a criação de interface visual customizada (VOGEL; MILINKOVICH, 2013).

Figura 8 – Arquitetura de um produto eclipse (RCP)



Fonte: (VOGEL; MILINKOVICH, 2013)

Ainda segundo os autores, um *plugin* eclipse, ou um produto RCP, apresentam dois arquivos de configuração importantes, são eles: *MANIFEST.MF*, que contém as informações necessárias para seguir a especificação OSGI, utilizada pelo Equinox; e o arquivo *plugin.xml*, que provê a possibilidade de criar e contribuir com elementos através da API da plataforma Eclipse, possibilitando sua extensão.

Com isso o desenvolvimento de um *plugin* para Eclipse 4 e de um produto RCP, são muito similares, a principal diferença está na forma como são empacotados. Quando o desenvolvimento é um *plugin*, este deve ser incorporado e instalado em um produto eclipse para ter as suas funcionalidades acessíveis, ao passo que em um produto RCP, é criada uma versão própria do eclipse trazendo apenas a base da plataforma mais os *plugins* desejados.

3.2 Processadores de Linguagem

Nessa seção apresentamos os processadores de linguagem utilizados para o processamento de linguagem natural e de linguagem formal. A subseção 3.2.1 apresenta a ferramenta Xtext, um produto Eclipse para a criação e processamento de linguagens formais; a subseção 3.2.2 apresenta o conjunto de ferramentas oferecidas pelo *Stanford Parser*, utilizado no PLN.

3.2.1 Eclipse Xtext

Xtext é um *framework* para o desenvolvimento de linguagens, que podem ser *Domain Specific Language* (DSL) ou linguagens de propósito geral. As linguagens desenvolvidas com Xtext podem ser facilmente integradas em IDEs baseadas na plataforma Eclipse (FOUNDATION, 2017b).

A Xtext se apresenta junto ao pacote "*eclipse DSL tools*", que oferece todos os recursos necessários para o desenvolvimento de uma nova linguagem. Seus recursos são facilmente acessíveis. Ela apresenta uma série de configurações genéricas que servem para a grande maioria dos casos, o que facilita muito sua utilização. Mesmo esse sendo um projeto mantido pela Eclipse Foundation, essa ferramenta oferece recursos que podem ser utilizados em qualquer ambiente Java através de APIs que descrevem os diferentes recursos da nova linguagem (FOUNDATION, 2017b), (BETTINI, 2016).

Ainda segundo os autores, a peça central da Xtext é a "*grammar language*", como o nome sugere, é a linguagem que utilizamos para especificar a gramática de uma linguagem, ou seja, é uma linguagem para especificação de linguagens. Por meio dela, é possível declarar as definições das regras terminais da linguagem, utilizando expressões regulares, necessárias para a análise léxica. Também define a sintaxe dessas regras terminais, informando sua ordem esperada, as quais podem ser agrupadas em regras não-terminais, ou regras intermediárias, o que caracteriza a gramática da linguagem. Assim as informações necessárias para as **análises Léxica e Sintática** podem ser apresentadas no mesmo arquivo.

A combinação dessas duas informações (léxica e sintática) com a representação do programa fonte em uma estrutura de árvore chamada AST, oferece os recursos necessários para realizar a posterior **análise semântica**. O modelo semântico da linguagem é gerado a partir da *grammar language*, e utiliza os recursos disponibilizados pelo EMF (ver seção 3.1). A Xtext cria e mantém um modelo EMF internamente sincronizado com a gramática. Para cada arquivo fonte que já passou pelo *parsing*, a Xtext cria uma instância desse modelo semântico em memória, e essa instância do modelo é por sua vez a AST do arquivo *parseado*. Em outras palavras, a gramática definida com a *grammar language* é convertida em um modelo EMF, que é basicamente um modelo de objetos capaz de funcionar em tempo de execução. Assim, conforme ocorre a análise léxica/sintática, a ferramenta vai instanciando os elementos desse modelo e fazendo as devidas associações em tempo de execução. Essa representação em memória possibilita que diferentes frentes criem ou editem esse modelo, isso pode ser útil no caso de *Domain Specific Modeling Languages* (DSMLs).

A **análise semântica** do arquivo fonte é realizada nesse modelo em memória mantido pelo EMF, que por sua vez é o AST. A Xtext oferece uma estrutura sua realização desta, chamada de *Validators*, onde podemos codificar as restrições impostas à linguagem, como por exemplo o *type-checking*. Algumas análises já são nativamente oferecidas pelo

framework Xtext, como a verificação de identificadores únicos para variáveis por exemplo (BETTINI, 2016).

Porém todas essas análises e artefatos fazem apenas a metade do caminho, que é garantir que o código fonte esteja sintática e semanticamente correto. A outra metade diz respeito a geração de um código intermediário, como pode ser visto na Figura 3 que ilustra as etapas de um compilador. No caso de linguagens de propósito geral, esse código intermediário é normalmente *assembly*, já em DSLs, esse código normalmente é o código fonte de outra linguagem de propósito geral, como Java por exemplo.

A geração de código também ocorre com base no modelo do AST em memória, a Xtext cria um arquivo contendo o esqueleto de um gerador de código, que apresenta em seu interior dois recursos básicos, um chamado de ***Resource***, que contém todo o AST em memória, e outro chamado ***IFIleSystemAccess***, que como o nome sugere, é utilizado para manipular o sistema de arquivos, onde vamos criar os arquivos objeto da nossa linguagem.

Os artefatos gerados pela Xtext apresentam recursos avançados de forma simples, alguns deles: ***Syntax highlighting***, colorindo palavras reservadas e comentários, ou sublinhando em vermelhos erros sintáticos; ***Error markers***, relacionado ao exemplo anterior, indicando erros diretamente no código fonte, como sublinhar em vermelho; ***Content assist***, o famoso ctrl+espaço, apresenta sugestões de código ou opção de *auto-complete*, dependendo do contexto; ***Hyperlinking e Hovering***, possibilita saltar para a declaração de algum bloco ou variável (ctrl+clique = *hyperlinking*) ou apenas parar o cursor do mouse sobre um elemento, para que um *pop-up* apresente informações sobre o elemento em questão (*hovering*); ***Quickfixes***, podemos codificar soluções rápidas para problemas comuns e por fim o ***Outline***, que apresenta o AST do arquivo atual e possibilita navegar pelo código através dele.

Os recursos produzidos pela Xtext podem ser combinados com os já apresentados na seção 3.1, com isso é possível a distribuição destes recursos via *plugin*, ou mesmo a criação de um produto eclipse específico para a nova linguagem. Finalizando, por se tratar de um projeto Eclipse, os artefatos produzidos pela Xtext são mais facilmente integrados à essa plataforma, porém existe também a possibilidade de integração com IntelliJ IDEA da JetBrains.

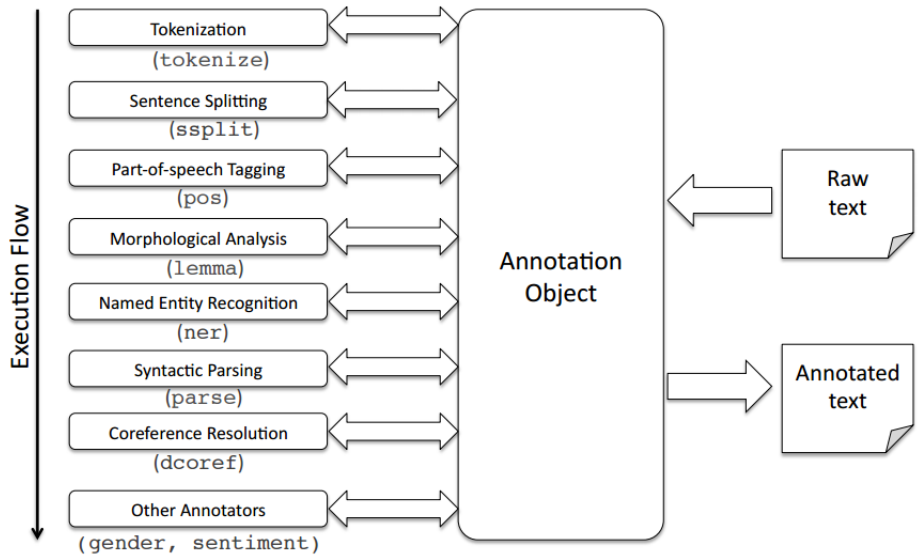
3.2.2 Stanford Parser

O conjunto de ferramentas oferecidas pelo grupo "*The Stanford Natural Language Processing Group*" é chamado de *Stanford CoreNLP* e oferece recursos avançados na área de PLN. Ele inclui ferramentas de segmentação (*tokenization*), *part-of-speech tagger*, *constituency parsing*, *dependency parsing* e várias outras ferramentas integradas em um único *framework*. Apresenta-se como uma biblioteca escrita na linguagem Java, e está disponível para *download* gratuitamente para vários idiomas, dentre eles podemos citar

o inglês, chinês, alemão, francês e espanhol. O português não é nativamente suportado por essa ferramenta, e além disso, a única língua que tem suporte total em termos de funcionalidades é a língua inglesa (MANNING et al., 2014).

O objetivo desse *framework* é fazer com que seja fácil aplicar um conjunto de ferramentas de análise linguística sobre textos, de forma flexível e extensível, e consiste em uma série de softwares independentes organizados em uma arquitetura do tipo *pipeline*, utilizada em sistemas que processam cadeias de dados, onde a saída de uma camada serve como insumo para outra. A Figura 9 apresenta o modelo da arquitetura utilizada pelo *Stanford CoreNLP*, onde *Raw text* é a entrada e *Annotated text* é a saída, sendo que a saída vai depender de qual "*Annotator*" foi requisitado.

Figura 9 – Visão geral da arquitetura do *Stanford CoreNLP*.



Fonte: (MANNING et al., 2014)

Observe que os processos são dependentes, de modo que todos necessariamente precisam iniciar pelo *Tokenization*, porém a saída vai ser definida em cascata até a ação requisitada, por exemplo, se a ação requisitada ao *Stanford coreNLP* for *Part-of-speech Tagging*, os processos *Tokenization* e *Sentence Splitting* são realizados, para só então executar o *tagging* e retornar o resultado. Esse tipo de arquitetura, onde uma etapa depende da saída da etapa anterior é chamado de *pipeline* (MANNING et al., 2014).

Essa arquitetura foi desenvolvida para atender um conjunto de objetivos listados a seguir:

- possibilitar a rápida e simples obtenção de anotações linguísticas para um texto;
- esconder a complexidade atrás de uma API única;

- facilitar o uso da ferramenta, tornando-a fácil de aprender e usar;
- prover um *framework* leve baseado em objetos Java.

Outras ferramentas oferecidas pelo *The Stanford Natural language Processing Group* que merecem destaque são a *Tregex* e a *Semgrex*. A *Treegex* consiste em uma ferramenta desenvolvida especificamente para o problema das buscas em árvore (*syntatic parse tree*). Ela oferece um mecanismo para encontrar padrões em estruturas de árvores anotadas sintaticamente, baseada em outra ferramenta de mesmo propósito (*TGrep2*), a *Tregex* apresenta funcionalidades de busca que funcionam de forma semelhante as expressões regulares para realizar suas buscas, eliminando a necessidade de implementação manual de algoritmos de busca em árvore, que são geradas pelo *constituency parsing*. A *Semgrex* é parecida, oferecendo recursos equivalentes, utilizando um mecanismo para encontrar padrões também baseado em expressões regulares, porém para a busca em grafos orientados, resultantes do *dependecy parsing* (LEVY; ANDREW, 2006).

3.3 Linguagem Relax

Whittle et al. (2010) desenvolveram uma linguagem para a especificação de requisitos para SAs chamada **RELAX**, de modo a endereçar os fatores de incerteza e características dos SAs. A linguagem RELAX se apresenta na forma de uma linguagem natural estruturada, baseada no vocabulário da língua inglesa, oferece operadores com valor semântico associado e estruturados, o que facilita o processamento computacional desses requisitos.

Os operadores RELAX possibilitam endereçar os fatores de incerteza, organizando o requisito em expressões mais formalizadas, porém com trechos de linguagem natural em seu interior, classificamos portanto como semi-formal. Vejamos um exemplo simples da utilização da RELAX retirado de Whittle et al. (2010):

Requisito em linguagem natural:

"*The fridge shall detect and comunicate with food package*".

Linguagem RELAX:

"*The fridge SHALL detect and communicate information with AS MANY food packages AS POSSIBLE*".

Expressão gramatical RELAX:

"*SHALL (AS MANY AS POSSIBLE p)*".

E por fim a expressão formal RELAX:

$\mathrm{AF}(\Delta(p) \in S)$.

Através dessa expressão, podemos inferir que a ação *detect and comunicate* ocorre no presente, e que a identificação dos *food packages* pode ser flexível, essa análise só é possível graças a semântica dos operadores, e a expressão gramatical gerada oferece uma síntese dessas semânticas presentes em um requisito. A expressão formal gerada

oferece toda a semântica para as expressões gramaticais geradas no processo de conversão. Nesse exemplo um quantificador foi definido (representado pela sigla AF), a proposição (símbolo "p") é um parâmetro para Δ, como essa proposição é variável, essa variação possui uma relação de pertinência com um conjunto *Fuzzy* (representado pela letra S). Porém muitas informações estão contidas em "p", e "p" continua expresso em linguagem natural, portanto a utilização da RELAX auxilia, mas ainda não possibilita a extração das informações necessárias aos modelos conceituais.

3.3.1 Vocabulário da Linguagem

O vocabulário da RELAX pode ser dividido em três partes, operadores modais (geralmente indicam ação), operadores temporais (usados para tratar as incertezas em termos da lógica *Fuzzy*) e os fatores de incerteza (comumente o ambiente onde o sistema está inserido).

Tabela 1 – Operadores Relax.

Operador Relax	**Descrição**
Operadores Modais	
SHALL	O requisito deve conter.
MAY..OR	O requisito especifica uma ou mais alternativas.
Operadores Temporais	
EVENTUALLY	O requisito deve conter eventualmente.
UNTIL	O requisito deve garantir até uma posição futura.
BEFORE, AFTER	O requisito deve realizar antes ou depois de um determinado evento.
IN	O requisito deve realizar durante um determinado tempo.
AS EARLY, LATE AS POSSIBLE	Um requisito especifica algo que deve realizar, logo ou deve se atrasar o maior tempo possível.
AS CLOSE AS POSSIBLE TO [frequência]	Um requisito especifica algo que acontece repetidamente, mas a frequência pode ser flexibilizada.
Operadores Ordinais	
AS CLOSE AS POSSIBLE TO [quantidade]	Um requisito especifica algo que acontece repetidamente, mas a quantidade pode ser flexibilizada.
AS MANY, FEW AS POSSIBLE	O requisito especifica uma quantidade contável, mas o número exato pode ser flexibilizado.
Fatores de Incerteza	
ENV	Define um conjunto de propriedades que compreendem o ambiente do sistema.
MON	Define um conjunto de propriedades que podem ser monitoradas através do sistema.
REL	Define a relação entre as propriedades **ENV** e **MON**.
DEP	Identifica as dependências entre os requisitos (relaxados e invariantes).

Fonte: (WHITTLE et al., 2010)

A sintaxe da linguagem é definida por meio de sua **gramática**, a qual define as construções permitidas com a utilização dos operadores apresentados na Tabela 1 mais

seus operandos em LN. As regras de produção são apresentadas na Figura 10.

Figura 10 – Gramática da RELAX.

ϕ : = true | false | p | SHALL ϕ
| MAY ϕ_1 OR MAY ϕ_2
| EVENTUALLY ϕ
| ϕ_1 UNTIL ϕ_2
| BEFORE e ϕ
| AFTER e ϕ
| IN t ϕ
| AS CLOSE AS POSSIBLE TO f ϕ
| AS CLOSE AS POSSIBLE TO q ϕ
| AS {EARLY, LATE, MANY, FEW} AS POSSIBLE ϕ

Fonte: (WHITTLE et al., 2010)

A gramática produz alguns símbolos que podem conter LN, como "e", que indica a ocorrência de um evento, "f" que indica uma frequência (em que um evento ocorre por exemplo), "q" que indica uma quantidade, "t" que indica um tempo e o já conhecido "p" que é uma proposição, onde normalmente se encontra a ação e o sujeito. Assim esses operadores, apesar de ainda estarem em LN, possuem uma semântica já definida, o que auxilia possíveis tarefas de PLN.

A semântica da linguagem está baseada na *Fuzzy Branching Temporal Logic* (FBTL), esse formalismo por sua vez está baseado na lógica *Fuzzy*, e permite descrever os modelos de ramificação temporal com informações temporais e lógicas. Isso possibilita capturar e representar as incertezas no requisito. A semântica da RELAX é apresentada na Tabela 2

3.4 Lições do Capítulo

Ferramentas importantes foram apresentadas nesse capítulo, dentre elas destacamos a Xtext, que torna o processo de desenvolvimento de uma nova linguagem algo simples, oferecendo inclusive todo o ferramental de apoio a nova linguagem, como IDE e gerador de código.

Outra ferramenta que merece nosso destaque é o *Stanford CoreNLP*, que é distribuído como uma biblioteca Java, possibilitando a utilização de recursos avançados de PLN de forma simplificada porém poderosa.

Por fim, a linguagem RELAX, que oferece recursos interessantes para a representação de incertezas em requisitos para SAs ela apresenta uma semântica bem definida, e combina elementos de linguagens formais com operandos em LNs.

Tabela 2 – Semântica das Expressões RELAX.

Expressão Relax	**Informal**	**Formalização FBTL**
SHALL ϕ	ϕ é verdade em qualquer estado	**AG** ϕ
MAY ϕ_1 *OR MAY* ϕ_2	em qualquer estado, ou ϕ_1 ou ϕ_2 são verdades	**AG** $(\phi_1 or \phi_1)$
EVENTUALLY ϕ	será verdade em algum estado futuro	**AF** ϕ
ϕ_1 *U* ϕ_2	ϕ_1 será verdadeiro até ϕ_2 tornar-se verdadeiro	**AF** ϕ
BEFORE e ϕ	ϕ é verdade em qualquer estado ocorrendo antes do evento **e**	**A** $\chi_<$ e_d é a duração até a próxima ocorrência de **e**
AFTER e ϕ	ϕ é verdade em qualquer estado que ocorre após o evento **e**	**A** $\chi_>$ e_d ϕ
IN t ϕ	ϕ é verdade em todo o estado no intervalo de tempo **t**	(*AFTER* t_start ϕ and *BEFORE* t_end ϕ) onde t_start, t_end são eventos que denotam o início e o fim do intervalo **t**, respectivamente
AS EARLY AS POSSIBLE ϕ	ϕ torna-se verdadeiro em algum estado tão perto do tempo atual quanto possível	**A** $\chi_>$ $=_d$ ϕ onde d é a duração difusa definida de tal forma que seus membros de função tem o seu máximo em 0 (ou seja, M(0)= 1) e diminui continuamente para os valores > 0
AS LATE AS POSSIBLE ϕ	ϕ torna-se verdadeiro em algum estado tão próximo do tempo t = ∞ possível	**A** $\chi_>$ $=_d$ ϕ onde d é a duração difusa definida de tal forma que seus membros de função tem o seu valor mínimo em 0 (ou seja, M(0)= 0) e aumenta continuamente para os valores >0
AS CLOSE AS POSSIBLE TO f ϕ	ϕ é verdade em intervalos periódicos, onde o período é tão perto de f possível	**A** ($\chi =_d \phi$ and $\chi = 2_d$ ϕ and $\chi = 3_d$ ϕ and...) onde **d** é a duração difusa definida de tal modo que a sua função de adesão tem o seu valor máximo, no período definido por f (ou seja, M(d)= M(2d)=...=1) e diminui continuamente para valores inferiores e superiores a d (e 2d, ...)
AS CLOSE AS POSSIBLE TO q ϕ	existe alguma função Δ tal que $\Delta(\phi)$ são quantificáveis e $\Delta(\phi)$ é tão próximo de 0 possível	**AF**(($\Delta(\phi)$-q) pertence a S, onde S é um conjunto fuzzy cuja função de composição tem um valor no zero (M(0)= 1) e diminui continuamente em torno de zero. $\Delta(\phi)$ "conta"o quantificável, que será comparado com q
AS MANY AS POSSIBLE ϕ	existe alguma função Δ tal que $\Delta(\phi)$ é tão perto de ∞ possível	**AF** ($\Delta(\phi)$ pertence a S), onde S é um conjunto fuzzy cuja função de pertinência tem 0 valor no zero (M(0)= 0) e aumenta continuamente em torno de zero
AS FEW AS POSSIBLE ϕ	existe alguma função Δ tal que $\Delta(\phi)$ é quantificáveis e é o mais próximo possível a 0	**AF** ($\Delta(\phi)$ pertence a S), onde S é um conjunto fuzzy cuja função de pertinência tem um valor no zero (M(0)= 1) e diminui continuamente em torno de zero

Fonte: (WHITTLE et al., 2010)

4 TRABALHOS RELACIONADOS

Nesse capítulo apresentamos os trabalhos relacionados à criação de modelos a partir de requisitos de software, obtidos através de um mapeamento sistemático da literatura baseado no guia proposto por Petersen et al. (2008). A seção 4.1 apresenta o de forma resumida o método utilizado para a realização do mapeamento. Na seção 4.2 são apresentados os resultados desse mapeamento sistemático, trazendo informações sobre os artigos publicados e suas contribuições mais relevantes e por fim na seção 4.3 apresentamos algumas lições aprendidas com a realização deste mapeamento.

4.1 Protocolo de Mapeamento

O objetivo do mapeamento é estudar o estado da arte da extração de modelos conceituais a partir de requisitos textuais de software. Primeiramente buscamos por trabalhos aplicados no cenário de SAs, porém um estudo inicial e exploratório logo mostrou que essa é uma área pouco explorada e portanto não há trabalhos suficientes, assim ajustamos nossas buscas incluir softwares em geral.

Partindo do objetivo geral desse trabalho, que é desenvolver uma ferramenta capaz de extrair modelos conceituais de requisitos de software para SAs, buscamos encontrar nesses trabalhos informações que nos auxiliem em direção a esse objetivo, para tal, formulamos as seguintes questões de busca:

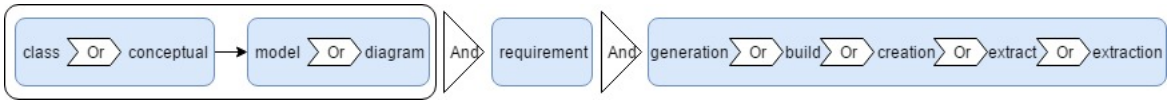
- Quais as tecnologias e técnicas necessárias para desenvolver uma ferramenta de extração de modelos/diagramas a partir de requisitos?
- Em que formato os requisitos devem estar apresentados?
- Como capturar conceitos dos requisitos?

O próximo passo do processo é criar a *string* de busca que deve ser usada como entrada em mecanismos de busca automática. Essa *string* deve ser robusta para pegar resultados com títulos e conteúdos semanticamente parecidos. Além disso definimos em qual/quais bases científicas as buscas devem ocorrer.

Para a realização desse trabalho, a base de buscas escolhida foi a *scopus*, por ser amplamente usada no meio acadêmico e agir como um indexador, tendo sob sua abrangência várias bases científicas. Na sequência é apresentada a estrutura da nossa *string* genérica.

Ao executarmos a *string*, retornaram 707 resultados, o que indica que a busca está muito abrangente, assim mais alguns filtros foram adicionados, sendo eles: Publicações realizadas entre 2011 e 2015; *Subject Area* = “Computer Science”; *Document Type* = “Conference paper” e “Article”; *Source Type* = “Journals” e “Conference Proceedings”; e por fim *Language* = “english”. Com esses limitadores, chegamos ao número de 138 artigos, que são a base dos passos posteriores.

Figura 11 – Estrutura genérica da *string* de buscas.



Fonte: O autor

Na sequência, com o objetivo de eliminar falsos positivos e encontrar trabalhos relevantes para a pesquisa, refinamos esse número utilizando de critérios de aceitação e exclusão listados a seguir:

Critérios de inclusão:

- Artigos que apresentam uma técnica ou ferramenta sobre a extração automática de modelos de software (como modelo conceitual), que automatizam alguma parte do processo de desenvolvimento ou trazem indicadores através da análise e processamento de requisitos de software.
- Artigos que apresentem alguma técnica ou ferramenta para uma especificação de requisitos que possibilite uma posterior análise ou processamento computacional sobre os mesmos.

Critérios de exclusão:

- Outros estudos secundários.
- Menos de 6 páginas.
- Artigos indisponíveis.

E por fim, o último passo é a leitura e classificação dos artigos. Essa etapa é realizada em dois passos, primeiramente ocorre a leitura do *abstract*, em busca de palavras chave que indiquem sua relevância. Ao realizar a leitura do *abstract*, do título e de suas palavras-chave, deve-se confrontá-los aos critérios de aceitação e exclusão, de modo a aceitar ou não tal trabalho. Nessa etapa, podem ocorrer alterações em tais critérios. Caso o *abstract* não seja suficiente, o revisor pode optar por ler a introdução e a conclusão para atestar o grau de relevância de um trabalho para essa pesquisa.

4.2 Resultados

Dos 138 artigos revisados, 3 estavam indisponíveis, e 13 foram selecionados e apresentam conteúdo relacionado ao tema sob pesquisa, e são abordados nas próximas seções. Na sequência apresentamos a Tabela 3 listando os trabalhos selecionados.

Tabela 3 – Trabalhos selecionados.

Título	**Autor**	**Ano**
From natural language requirements to UML class diagrams	Sharma, Srivastava e Biswas (2015)	2015
An automated object-based approach to transforming requirements to class diagrams	Dahhane et al. (2014)	2014
aToucan: An automated framework to derive UML analysis models from use case models	Yue, Briand e Labiche (2015)	2015
Conceptual modeling of natural language functional requirements	Sagar e Abirami (2014)	2014
Eliciting relations from natural language requirements documents based on linguistic and statistical analysis	Liu, Li e Kou (2014)	2014
RSL-IL: An interlingua for formally documenting requirements	Ferreira e Silva (2013)	2013
Towards Generation of Sequence Diagrams from Operation Contracts and Design Patterns	Shakya e Nantajeewarawat (2013)	2013
Domain ontology based class diagram generation from functional requirements	Jyothilakshmi e Samuel (2012)	2012
Software Requirements Translation from Natural Language to Object-Oriented Model	Fatwanto (2012)	2012
A knowledge-based system for improving the consistency between object models and use case narratives	Bolloju, Schneider e Sugumaran (2012)	2012
SBVR2UML: A challenging transformation	Afreen, Bajwa e Bordbar (2011)	2011
Parsed use case descriptions as a basis for object-oriented class model generation	Elbendak, Vickers e Rossiter (2011)	2011
Semi-automatic generation of UML models from natural language requirements	Deeptimahanti e Sanyal (2011)	2011

As próximas seções apresentam nossas questões de busca, e as respostas para elas obtidas por meio do estudo nos trabalhos selecionados.

4.2.1 Quais as tecnologias e técnicas necessárias para realizar uma ferramenta de extração de modelos/diagramas a partir de requisitos?

A maioria das abordagens optou pelo uso de técnicas de PLN, com oito ocorrências. Propostas de especificações formais (EF) e abordagens híbridas (HB), que combinam o uso de PLN com ontologias, apresentam duas ocorrências cada, e há apenas uma abordagem totalmente dirigida por ontologias (ONT), como indica a coluna Técnicas/Contribuições da Tabela 4. Porém essa classificação é superficial, pois alguns dos trabalhos que são classificados em PLN, utilizam princípios formais, oferecendo regras para a especificação de requisitos (para diminuir a ambiguidade).

Respondendo a questão, as duas principais técnicas para possibilitar a extração dos conceitos de requisitos é a formalização (ou semi-formalização) da especificação de requisitos e PLN, em muitos casos ocorre a combinação delas, evidenciamos que o maior desafio das abordagem que utilizam especificações de requisitos descritos em LN é justamente encontrar algum formalismo em suas sentenças. Para tal, uma ferramenta largamente

Tabela 4 – Técnicas usadas pelos autores.

Autor	**Requisitos**	**Técnica/Contribuição**
(SHARMA; SRIVASTAVA; BISWAS, 2015)	LN	PLN
(DAHHANE et al., 2014)	SFL	PLN
(YUE; BRIAND; LABICHE, 2015)	SFL	PLN
(SAGAR; ABIRAMI, 2014)	LN	PLN
(LIU; LI; KOU, 2014)	LN	PLN
(FERREIRA; SILVA, 2013)	FL	EF
(SHAKYA; NANTAJEEWARAWAT, 2013)	SFL	ONT
(JYOTHILAKSHMI; SAMUEL, 2012)	LN	HB
(FATWANTO, 2012)	FL	EF
(BOLLOJU; SCHNEIDER; SUGUMARAN, 2012)	LN	HB
(AFREEN; BAJWA; BORDBAR, 2011)	SFL	PLN
(ELBENDAK; VICKERS; ROSSITER, 2011)	LN	PLN
(DEEPTIMAHANTI; SANYAL, 2011)	LN	PLN

Legendas:
LN = Linguagem Natural
FL = Formal
SFL = Semi Formal
PLN = Processamento de Linguagem Natural
EF = Especificação Formal
ONT = Ontologias
HB = Híbrido

usada é o *Stanford Parser*. Além disso, outras tecnologias que podemos citar são: *Protegé* para a modelagem de Ontologias e suas regras, a ferramenta *Computer-Aided Software Engineering* (CASE) ArgoUML, que é *open-source*, o que permite a adição de funcionalidades, a linguagem Java, que foi usada na maioria dos trabalhos e apresenta uma série de bibliotecas que auxiliam tanto em técnicas PLN quanto no uso de ontologias, além da plataforma Eclipse, que possibilita a criação de *plugins* aproveitando seus recursos oferecidos. Os próximos parágrafos detalham essa resposta.

A abordagem proposta por Sharma, Srivastava e Biswas (2015) propõe uma ferramenta para a criação de diagramas a partir de requisitos descritos em LN, para isso eles utilizam técnicas de PLN com o auxílio da ferramenta *Stanford Parser*. O *parser* atua extraindo a estrutura sintática de cada sentença, assim os autores levantaram um conjunto de estruturas sintáticas mais comumente encontradas em especificações de requisitos, chamados *Grammatical Knowledge Pattern* (GKP), e os comparam com a estrutura extraída dos requisitos, buscando identificar qual o GKP da sentença. Cada GKP possui pontos chave, que são mapeados para elementos no modelo conceitual de classes.

Dahhane et al. (2014) usa técnicas de PLN sobre requisitos de software descritos no formato de histórias de usuário contendo um conjunto de restrições definidas pelos autores, expressas em regras de escrita. Esse conjunto de regras atua como uma estrutura sintática, e com ela então a extração dos elementos do modelo dessa história de usuário é realizada.

Já Yue, Briand e Labiche (2015) apresentam uma ferramenta promissora (aToucan), que utiliza técnicas de PLN combinadas, processando requisitos de software descritos na forma de casos de uso restritos (*Restricted Use Case Modeling*). Os autores definiram

26 regras para ordenar a escrita em LN para esses casos de uso. O *Stanford Parser* é usado como ferramenta de processamento para gerar o que os autores chamam de *Use Case Metamodel* (UCMeta), que é uma linguagem intermediária carregada de semântica computável que possibilita a extração dos elementos conceituais. A ferramenta foi construída como um *plugin* para a plataforma Eclipse, e para desenhar os modelos utilizou um recurso chamado *Kermeta*, também oferecido pela plataforma.

A abordagem proposta por Sagar e Abirami (2014) atua sobre requisitos de software descritos em LN na língua inglesa. Para realizar o PLN o *Stanford Parser* é novamente usado, de modo a extrair a estrutura sintática das sentenças, também chamada de gramática, com isso em mãos é possível realizar a extração dos elementos do modelo conceitual através de regras de mapeamento baseadas nas relações gramaticais das palavras e sentenças.

Os autores Liu, Li e Kou (2014) buscam criar um modelo de dependência estratégica para os requisitos descritos em chinês usando o *framework i**. Novamente a técnica aplicada é PLN, de modo a obter a estrutura sintática das sentenças, carregando tais sentenças com a semântica de suas partes, para na sequência identificar e marcar as entidades com *tags* e por fim fazer o mapeamento entre os requisitos, proposto por esse modelo no contexto do *framework i**.

A técnica utilizada por Ferreira e Silva (2013) consiste em uma linguagem formal para a especificação de requisitos, que é abordada adiante. O maior mérito do trabalho é ilustrar, apesar de não implementar, como deve ser feito o mapeamento entre os elementos da especificação de requisitos para um modelo.

Uma abordagem diferente é proposta por Shakya e Nantajeewarawat (2013), onde a principal tecnologia utilizada é a ontologia. O objetivo deste trabalho é a criação de diagramas comportamentais (diagrama de sequência) para padrões de projetos identificados nos requisitos. Para possibilitar o processamento computacional dos padrões de projeto, os mesmos foram modelados em *Ontology Web Language* (OWL), e as solução providas por tais padrões foram especificadas na forma de *Semantic Web Rule Language* (SWRL) usando a ferramenta *protegé*. Além disso o diagrama de classes e os casos de uso com seus devidos *Operation Contracts* devem também ser traduzidos para OWL e SWRL respectivamente para cada instância do problema, para só então serem processados e gerar o diagrama de sequência, ou seja, todo o raciocínio ocorre sobre as ontologias e suas regras.

Já a abordagem proposta por Jyothilakshmi e Samuel (2012) é híbrida, combinando técnicas de PLN com ontologias. Ela busca criar um modelo conceitual a partir de requisitos descritos em LN. A abordagem exige um modelo de ontologia em OWL específico para o domínio sob análise, servindo como base de conhecimento para obtenção dos elementos conceituais. Já o PLN busca substantivos e verbos nas especificações realizando o *parsing* e o *tagging* (porém não informa qual ferramenta é usada pra isso) e compara-os com a ontologia de domínio específico para identificar os elementos do modelo

conceitual. Para modelagem de ontologias, os autores recomendam a ferramenta *protegé*.

A principal contribuição de Fatwanto (2012) está no uso de técnicas formais, seu trabalho apresenta uma linguagem formal para especificar requisitos, mas não gera nenhum diagrama ou modelo, porém seu trabalho contribui no sentido de possibilitar o processamento computacional desses requisitos, ilustrando um conjunto de regras para transcrever suas especificações em modelos UML.

O trabalho de Bolloju, Schneider e Sugumaran (2012) apresenta novamente uma abordagem híbrida, combinando o uso de ontologia com PLN. Os autores buscam realizar a verificação automática dos modelos, apresentando recomendações aos analistas sobre problemas encontrados em seus modelos, isso apresentado como uma extensão para a ferramenta CASE ArgoUML. Para isso o PLN foi usado para identificar padrões nas especificações que possivelmente serão usados em um modelo conceitual, uma análise é então realizada sobre essas informações com o auxílio de ontologias e comparadas com o estado atual dos modelos, gerando assim as recomendações. Uma série de ferramentas *open-source* foram utilizadas (além da ArgoUML), *protegé* para as ontologias, e bibliotecas Java para possibilitar o processamento dessas ontologias como *Jess* e *JessTab*.

Os autores Afreen, Bajwa e Bordbar (2011) utilizam técnicas de PLN sobre requisitos estruturados, descritos no padrão *Semantic of Business Vocabulary and Rules* (SBVR) de modo a obter um modelo conceitual. Para realizar o PLN a ferramenta *Stanford Parser* foi novamente usada para obter os conceitos, tais conceitos são submetidos a um conjunto de regras de extração para definir quais elementos fazem parte do modelo conceitual, para posteriormente o desenhar com a ajuda de funções gráficas da linguagem Java, como *drawRect()*, *drawLine()*.

Com Elbendak, Vickers e Rossiter (2011) novamente uma abordagem baseada em PLN é utilizada para processar requisitos descritos em LN de modo a identificar os elementos de um modelo conceitual. Novamente um *parser* é usado, porém os autores não deixam claro qual, para encontrar a gramática e a partir dela criar uma linguagem intermediária com valor semântico associado às sentenças, para posterior extração e identificação dos conceitos através de um conjunto de regras definidas pelos autores.

E por fim a abordagem de Deeptimahanti e Sanyal (2011) trabalha também com requisitos descritos em LN e utiliza portanto PLN. Primeiramente realiza uma normalização nos requisitos, reescrevendo-os de modo a torná-los processáveis computacionalmente, utilizando ferramentas como *Stanfod Parser*, *JavaRAP* e *WordNet2.1* no processo, na sequência a ferramenta desenvolvida é capaz de extrair diagramas de casos de usos, de comunicação e modelo conceitual de classes.

4.2.2 Em que formato os requisitos devem estar apresentados?

Os trabalhos selecionados podem ser divididos em duas formas de expressar os requisitos, em sentenças em LN, comumente encontrados em descrições de casos de uso e

cenários, ou expressos de maneira formal ou semi-formal, apresentando um conjunto de regras de modo a restringir e diminuir os problemas mais comumente encontrados em PLN como ambiguidade por exemplo. A coluna Requisitos da Tabela 4 exibe que o formato mais usado para especificação de requisitos nos trabalhos relacionados é a LN, com sete ocorrências, ao passo que abordagens formais ou semi-formais somam seis ocorrências, mais detalhes serão apresentados na sequência.

Linguagem natural:

Na abordagem de Bolloju, Schneider e Sugumaran (2012) foi desenvolvida uma extensão da ferramenta CASE ArgoUML. Ela oferece uma interface para combinar os diagramas de casos de uso com suas narrativas, descritas em inglês. A combinação do diagrama com suas narrativas são a entrada para os passos posteriores. Os autores Elbendak, Vickers e Rossiter (2011) buscavam fazer o processamento também baseados em casos de uso descritos em inglês, porém devido a natureza aberta e livre da LN, usada nas descrições dos casos de uso, optaram por aceitar qualquer documento de especificação de software descritos em LN, pois os princípios são os mesmos.

Os autores Sagar e Abirami (2014), Deeptimahanti e Sanyal (2011), Sharma, Srivastava e Biswas (2015) e Jyothilakshmi e Samuel (2012) também realizam a extração dos conceitos de especificações de requisitos expressas em LN, todos esses trabalhos apresentaram pelo menos um caso de estudo onde a entrada foi um cenário descrito em inglês, contendo vários requisitos funcionais.

E por fim Liu, Li e Kou (2014) que processa requisitos descritos em LN na língua chinesa.

Formal ou semi-formal

A ferramenta proposta por Yue, Briand e Labiche (2015) utiliza uma descrição restrita de casos de uso, chamada *Restricted Use Case Modeling* (RUCM), apresentando 26 regras descritivas que segundo os autores diminui a ambiguidade das especificações, o que classifica essa abordagem como Semi-formal.

Afreen, Bajwa e Bordbar (2011) utilizam requisitos descritos no padrão *Semantics Of Business Vocabulary And Rules* (SBVR), que é uma recomendação da *Object Management Group* (OMG) para especificação de requisitos, essa especificação busca dar base formal para declarações em LN, combinando elementos formais com LN (caracterizando uma abordagem Semi-formal), facilitando a computação de seus requisitos e sua escrita por humanos.

No trabalho de Dahhane et al. (2014) histórias de usuário são usados para a especificação de requisitos, os autores desenvolveram uma série de regras na forma de restrições para sua escrita, baseados na *Object Constraint Language* (OCL) usado na UML, e criaram então as *Constraint Story Card Template* (CSC) que apresenta *templates* para expressar conceitos, relações e métodos. Os resultados apresentados não mencionam atributos. As restrições continuam agindo sobre trechos em LN, caracterizando essa abordagem como

semi-formal.

A abordagem de Ferreira e Silva (2013) propõem uma linguagem formal para a especificação de requisitos chamada RSL-IL. Os requisitos são descritos em uma linguagem natural restrita, com um conjunto de construtores para as sentenças, seguindo um *template* definido de modo a carregar semântica, atingindo assim o formalismo.

Fatwanto (2012) propôs uma linguagem formal, a abordagem proposta pelo autor é baseada em *Concern-Aware Requirements Engineering* (CARE). O *framework* CARE busca os interesses dos *stakeholders* e em seguida os transcreve para um requisito. Seguindo o formato : *Requirement* <- *Subject* + *Verb* + *Target* + [*Way*]; onde *Way* é opcional, e possui regras específicas em seu conteúdo para possibilitar a identificação de relacionamentos, além dessas, o autor apresenta um conjunto de regras para um possível mapeamento dos requisitos para um modelo conceitual.

Apesar de o trabalho de Shakya e Nantajeewarawat (2013) utilizar casos de uso descritos em LN inicialmente, ela exige contratos de operações para atuar em conjunto com cada caso de uso. Um contrato oferece detalhes e informações sobre o efeito de uma determinada operação realizada por um caso de uso. Além disso, nessa abordagem cada contrato, assim como os casos de uso, devem ser transcrito para OWL. Assim classificamos essa abordagem como semi-formal.

4.2.3 Como capturar conceitos dos requisitos?

Uma abordagem recorrente e mais encontrada nos trabalhos para a obtenção dos elementos do modelo conceitual, consiste na análise das sentenças dos requisitos, onde os autores possuem a estrutura sintática (*syntatic parse tree*) da sentença sob análise, de modo a identificar as dependências e as classes gramaticais das palavras (informando o que é verbo, sujeito e substantivo por exemplo), assim as abordagens apresentam regras de extração de tais elementos textuais para elementos conceituais, a principal diferença entre esses trabalhos está nessas regras, ou nas tecnologias usadas para representar essas regras. A obtenção da estrutura sintática (gramática) em algumas abordagens ocorre por meio tecnologias e técnicas de PLN, porém há abordagens que utilizam especificações de requisitos de modo formal ou semi-formal, dessa forma essa estrutura já é conhecida, focando apenas nas regras de extração. Há também algumas abordagens que apresentam apenas a linguagem formal para especificar requisitos, sem definir nenhuma regra de extração, e uma única abordagem que foca na obtenção de modelos comportamentais, o que exige o modelo conceitual como entrada e portanto não se aplica a essa questão.

Como a coluna Técnica/Contribuição da Tabela 4 ilustra, a grande maioria das abordagens utiliza técnicas de PLN para obter tais conceitos, mesmo quando a linguagem de especificação é formal ou semi-formal. As abordagens de Sagar e Abirami (2014), Yue, Briand e Labiche (2015), Afreen, Bajwa e Bordbar (2011), Deeptimahanti e Sanyal (2011) e Sharma, Srivastava e Biswas (2015) afirmam utilizar a ferramenta *Stanford Parser*,

para primeiramente quebrar a especificação do requisito em sentenças menores, chamadas *tokens*, por meio de elementos textuais como pontuação por exemplo. Na sequência os *tokens* são assinados por um *Part-Of-Speech* (POS) *tagger*, essa etapa vai indicar para cada *token* de uma sentença se ele é o sujeito, verbo, substantivo, etc. Depois disso é gerado o *syntatic parse tree* de cada sentença. Em alguns casos, os *tokens* são transformados em sua forma básica (em um processo chamado *stemming*).

Algumas abordagens criam uma linguagem intermediária usando as sentenças dos requisitos, acrescentando a cada elemento seu valor semântico através de sua classe gramatical ou POS, outras criam um glossário dos conceitos já encontrados, de modo a não gerar elementos redundantes nos modelos.

Com a estrutura sintática (*syntatic parse tree*) em mãos, as abordagens supracitadas, de forma sucinta, focam na análise dos substantivos, verbos, adjetivos e advérbios e através de um conjunto de regras de extração, definidas por eles, buscam encontrar a partir disso as classes, métodos/relacionamentos, atributos e relacionamentos respectivamente.

O trabalho de Elbendak, Vickers e Rossiter (2011) segue a mesma linha, porém não indica qual *parser* foi usado.

A abordagem de Bolloju, Schneider e Sugumaran (2012), as regras de extração (substantivo -> classes, etc) estão descritas em OWL, usando portanto uma abordagem hibrida entre PLN para obter os elementos e ontologias para modelar o conhecimento necessário para mapear tais elementos.

Os autores Liu, Li e Kou (2014) buscam encontrar dependências entre requisitos, assim os requisitos são processados de forma semelhante até encontrar as estruturas sintáticas das sentenças, depois disso, aplica um conjunto de regras próprias focadas na obtenção de relacionamentos de requisitos e não de conceitos para modelos conceituais.

Já Jyothilakshmi e Samuel (2012), depois de obter a estrutura sintática, analisam cada elemento das sentenças contra uma ontologia de domínio específico para o sistema sob desenvolvimento, comparando os elementos. Os elementos que são encontrados nessa ontologia são diretamente mapeados para o modelo, ao passo que os não encontrados são submetidos a um conjunto de regras baseados no mesmo princípio dos trabalhos anteriores, combinando assim duas técnicas.

Na abordagem proposta por Dahhane et al. (2014), um *template* para escrita de histórias de usuário é definido, de forma que esse *template* já oferece uma "estrutura sintática", assim regras de extração foram criadas seguindo os moldes anteriores.

Uma linguagem formal, é definida por uma gramática, e é isso que os autores Ferreira e Silva (2013) e Fatwanto (2012) propuseram. A maioria das abordagens anteriores faz um pré-processamento para justamente identificar essa gramática (estrutura sintática). Nesses dois casos, o trabalho foi justamente apresentar uma forma de especificar requisitos já com estrutura sintática definida. Assim a extração dos conceitos não foi apresentada, mas pode seguir os mesmos moldes dos trabalhos já citados.

E por fim Shakya e Nantajeewarawat (2013), apresentam a única abordagem que atua totalmente sobre ontologias, e seu objetivo é a criação de diagramas de sequência, nesse caso o diagrama de classes, com todos os conceitos deve ser apresentado como entrada, codificado em OWL, além das especificações de requisitos, e informações sobre padrões de projeto, todos codificados em OWL ou SWRL, desse modo o trabalho objetiva identificar os eventos e sua sequência, e não seus conceitos.

4.3 Lições do Capítulo

Como podemos observar, a área já é bastante explorada em termos de pesquisa quando não consideramos SAs, e notamos uma semelhança em várias abordagens, o que nos oferece uma base sólida e comumente usada, que é a combinação de técnicas de PLN com algum (semi-)formalismo na especificação de requisitos. Percebemos que a extração dos elementos conceituais exige que as sentenças dos requisitos possuam semântica computável, geralmente obtida por meio da estrutura sintática das sentenças, pontuamos que o maior desafio das abordagens que utilizam LN está justamente na obtenção dessa estrutura, o que indica que linguagens formais ou semi-formais auxiliam nesse sentido, facilitando e em alguns casos até eliminando essa etapa.

Ferramentas importantes foram apresentadas, dentre elas a mais comumente usada foi o *Stanfod Parser*, que oferece uma série de funcionalidades necessárias para o andamento do PLN com a língua inglesa, pontuamos que ele foi usado apenas em abordagens onde os requisitos estavam apresentados em LN ou de modo semi-formal, e sua principal aplicação nos trabalhos listados, é a obtenção da estrutura sintática das sentenças, reforçamos que essa etapa mostrou-se desnecessária quando os requisitos estavam expressos de maneira formal. Além do *Stanford Parser*, destacamos também as ontologias e a linguagem Java, que oferece uma série de bibliotecas tanto para uso com PLN quanto para ontologias.

Além disso, várias abordagens apresentam regras de extração dos elementos conceituais, focando suas análises principalmente sobre substantivos, adjetivos, verbos, advérbios e o sujeito das sentenças dos requisitos, que normalmente se tornam classes, atributos, métodos/relacionamentos. E finalizando, um padrão para a avaliação dos resultados de ferramentas com esse propósito pôde ser identificado nos trabalhos, o qual utilizamos no Capítulo 6 de resultados.

5 EXTRAÇÃO DOS CONCEITOS

Nesse capítulo descrevemos como ocorreu o desenvolvimento da nossa ferramenta de extração de modelos conceituais. A seção 5.1 apresenta uma visão geral da solução. A seção 5.2 apresenta os recursos que utilizamos para formular nossas regras de extração. A seção 5.3 apresenta a lista de regras de extração que desenvolvemos. Na seção 5.4 apresentamos nosso projeto e arquitetura do protótipo de ferramenta, além de detalhes de sua implementação. Na seção 5.5 apresentamos nossa solução desenvolvida e por fim a seção 5.6 aborda as lições aprendidas.

5.1 Visão Geral da Solução

A proposta de solução é baseada na criticidade da tarefa de modelagem, onde os analistas normalmente realizam tal atividade de modo manual, o que torna o processo suscetível a erros e lento. Além disso temos como hipótese não confirmada que as técnicas de modelagem utilizadas por agentes humanos normalmente seguem seu conhecimento empírico na atividade, o que torna seus resultados altamente dependentes do modelador. Mecanismos de automação dessa atividade atuam nessas três situações encontradas: o processo de extração de modelos conceituais utilizando uma ferramenta automatizada diminui os riscos de erros por omissão; é executado computacionalmente, logo é muito mais rápido que o processo manual; e por fim, impõem uma abordagem sistemática para a extração dos conceitos dos documentos de requisitos, diminuindo também o problema da dependência.

Nossa solução consiste em uma ferramenta que dê suporte a especificação de requisitos RELAX e que seja capaz de extrair os conceitos de um modelo conceitual de classes de forma automática, partindo do requisito textual e gerando uma lista de conceitos como classes e relacionamentos como saída.

Para tal solução, utilizamos especificações de requisitos expressos em inglês, pelos seguintes motivos: As melhores ferramentas relacionadas a PLN estão disponíveis para esse idioma; A linguagem de especificação de requisitos RELAX, é considerada semi-formal ou uma LN estruturada, baseada na língua inglesa.

5.2 Recursos Utilizados

Para formular nossas regras de extração, combinamos três elementos: a linguagem RELAX apresentada anteriormente (ver seção 3.3), que oferece um certo grau de formalismo com semântica associada, as vantagens de utilizarmos uma linguagem já estruturada são apresentadas no Capítulo 4, de trabalhos relacionados; assumimos que o modelo conceitual resultante é modelado com base em um *profile* UML específico apresentado na subseção 5.2.1, dese *profile* utilizamos sua estratégia de modelagem, que auxiliou nas extrações; e por fim, o uso de recursos e técnicas de PLN apresentados na subseção 5.2.2,

e as vantagens de sua utilização também foram apresentadas nos trabalhos relacionados. Assumimos algumas premissas que devem ser respeitadas para obter um melhor resultado do extrator:

1. os requisitos devem conter apenas uma ação;

2. os requisitos devem conter apenas um ator;

3. os requisitos devem conter apenas uma ocorrência para cada operador da RELAX, com exceção do operador AS CLOSE AS POSSIBLE TO, que pode receber parâmetros diferentes com semânticas diferentes, caracterizando dois operadores sobrepostos;

Além disso, nos baseamos nas diretrizes propostas por Cockburn (2005), que recomendam que a estrutura das sentenças dos requisitos devem ser o mais simples possível, com frases do tipo: Sujeito ... verbo ... objeto direto ... frase proposicional. Portanto o requisito deve deixar claro quem vai executar a ação, e qual ação será executada de acordo com a intenção do ator.

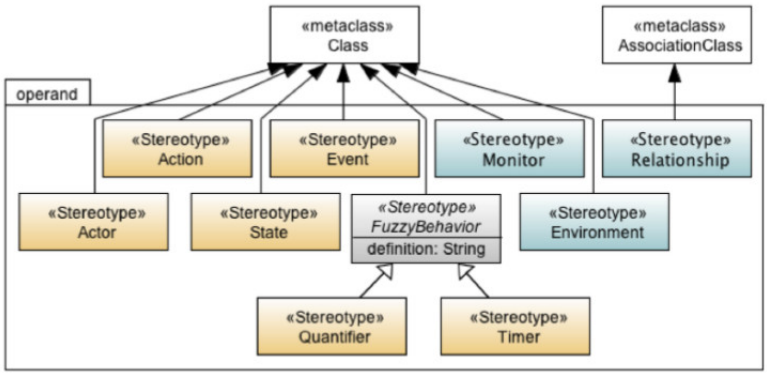
Nosso extrator tira do texto apenas informações explicitamente escritas, pois não há meios de inferir informações implícitas, como alguns relacionamentos por exemplo. Nesse ponto tiramos vantagem do *profile*, onde utilizamos suas regras e recomendações de modelagem como regras implícitas para a extração do nosso modelo. Além disso, codificamos algumas recomendações sugeridas pelos autores do *profile*, como nomenclatura de classes por exemplo.

5.2.1 *Profile* UML RelaxML

Nossa solução extrai conceitos para um modelo conceitual UML com um *profile* específico para modelagem de SAs, esse *profile* é um projeto em andamento do Laboratório de Engenharia de Software Aplicada (LESA) chamado de **RelaxML**. Seu desenvolvimento foi baseado na RELAX, e apresenta estereótipos para todos os seus operadores e operandos, proporcionando mais expressividade ao modelo conceitual. O meta modelo do *profile* foi dividido em dois pacotes, "operandos" na Figura 12 e "operadores" na Figura 13. Os operadores da RELAX são representados por meio de estereótipos no relacionamento de dependência, e os operandos por meio estereótipos nas classes, a semântica continua a mesma da linguagem RELAX.

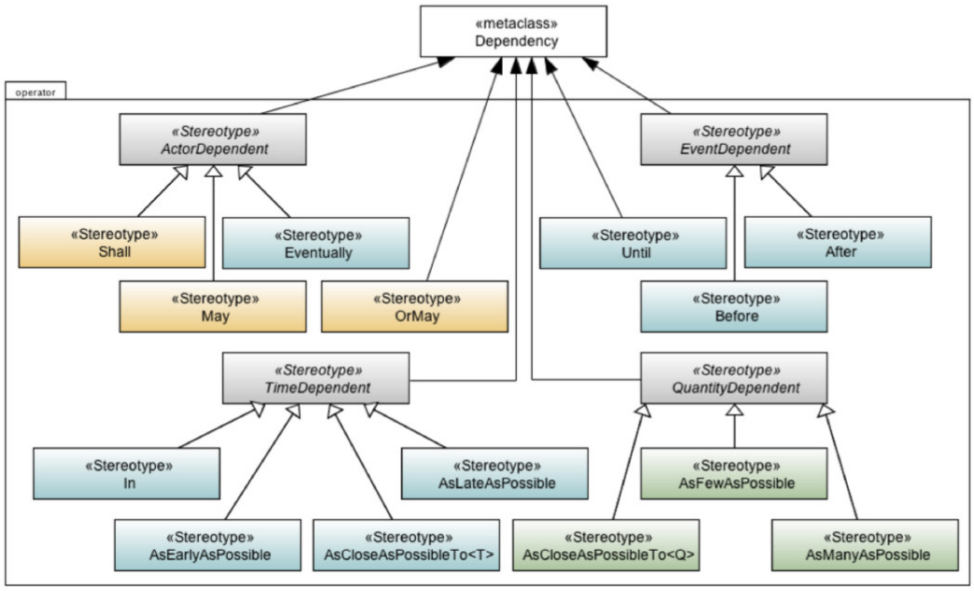
Além dos estereótipos, o *profile* apresenta regras do tipo *Object Constraint Language* (OCL), que acrescentam uma série de restrições ao modelo, que tornam o requisito facilmente visível no modelo conceitual. Nossas regras de extração devem também respeitar essas restrições, e em alguns casos, tirar proveito delas. As regras OCL do *profile* são:

Figura 12 – Operandos do *profile* RelaxML



Fonte: Grupo LESA

Figura 13 – Operadores do *profile* RelaxML



Fonte: Grupo LESA

1. Dependências com os estereótipos *Shall*, *May* e *Eventually* ligam classes do tipo *Actor* com *Action*;

2. Dependências com estereótipos *Before* e *After* ligam classes do tipo *Action* com *Event*;

3. Dependências com estereótipos *In, AsEarlyAsPossible, AsLateAsPossible* e *AsCloseAsPossible<T>* ligam classes do tipo *Action* com *Timer*;

4. Dependências com estereótipos *AsManyAsPossible, AsFewAsPossible* e *AsCloseAsPossible<Q>* ligam classes do tipo *Action* com *Quantifier*;

5. Dependências com estereótipo *OrMay* ligam dependências *May* com classes *Action*;

6. Dependências com o estereótipo *Until* ligam classes *Action* com *State*;

5.2.2 Processamento de Linguagem Natural

O núcleo e a parte mais desafiadora do nosso projeto foi o tratamento das sentenças em LN. Para isso utilizamos o conjunto de ferramentas do *Stanford CoreNLP*. O *parsing* utilizado foi o de dependências (ver subseção 2.4.4) por ser mais rápido e sua saída mais simples de tratar, mas ainda assim suficientes para nossos propósitos.

Para a extração dos dados, utilizamos o grafo orientado resultante do processo de *parsing* de dependências, representado pela classe *SemanticGraph*. Em nosso trabalho utilizamos duas abordagens distintas para percorrer o grafo de forma automática: a primeira para encontrar nós mais específicos e mais significativos, como o sujeito da frase por exemplo; a segunda para, a partir do sujeito encontrar componentes relevantes que estejam diretamente ligados a ele, como por exemplo o relacionamento "*compound*", que indica que um sujeito é composto por mais de uma palavra.

Para a primeira etapa, utilizamos a ferramenta ***Semgrex***, onde formulamos expressões para encontrar os nós mais significativos, pois suas expressões permitem criar regras mais elaboradas, um exemplo de expressão *semgrex* é apresentado a seguir:

```
{$} >>/.*subj/ {tag:/NN.*/}=sbj ?>>compound {}=sbjC
```

Essa expressão significa: a partir do nó raiz ({$}), procure em todos os descendentes (`>>`) qualquer relacionamento do tipo sujeito (/.*subj/) com um outro nó que seja um substantivo ({tag:/NN.*/}). Os caracteres "/" delimitam uma expressão que segue os mesmos padrões das expressões regulares, perceba que podemos incluir filtros não só para os relacionamentos, mas também para os nós. O sinal de atribuição "=sbj", cria uma âncora nos nós que casam com a expressão, possibilitando a posterior recuperação. Depois fazemos uma busca opcional nos nós descendentes da raiz (`?>>`,o "?" indica opcionalidade) procurando relacionamentos do tipo "*compound*", nesse caso não nos preocupamos com o conteúdo do nó. Finalizando adicionamos essa informação a uma âncora (sbjC).

Perceba que o segundo filtro de relacionamento (?>>*compound*) também atua sobre o nó raiz ({$}). Para esse segundo relacionamento atuar entre o segundo nó, ancorado com "sbj", e o terceiro nó, ancorado com "sbjC", teríamos de criar um "grupo" em torno deles, usando o parênteses para tal, da seguinte forma:

```
{$} >>/.*subj/ ({tag:/NN.*/}=sbj ?>>compound {}=sbjC)
```

Vejamos mais um exemplo de expressão *Semgrex*:

```
{$} >>/nmod|.*obj/ ({}=obj ?>>/amod|compound/ {pos:/NN.*/}=objC)
```

Procuramos descendentes do nó raiz que sejam ou o objeto da sentença, ou um modificador nominal (normalmente apresenta atributos), adicionalmente, procuramos compostos ou adjetivos ligados ao objeto modificador nominal, se houver. Finalizando, podemos negar qualquer expressão utilizando o operador "!", por exemplo:

```
{} !> {tag:/VB.*/}
```

Significa qualquer nó que seu descendente direto NÃO seja um verbo.

Para os nós encontrados com as expressões *Semgrex*, percorremo-os em um laço, verificando para cada ocorrência, relacionamentos mais simples, como composições ou modificadores nominais por exemplo, a segunda abordagem se deu via código Java, com métodos como *getChildWithReln(node, relation)* e *getChildrenWithReln(node, relation)*, que como o nome sugere, recuperam nós descendentes de um nó específico (primeiro parâmetro) que contenham determinado relacionamento (segundo parâmetro). O Código 5.1 apresenta um exemplo de código combinando as duas abordagens, para as classes do tipo *Actor*.

```
1  public ExtractedClass parseActor(String req) {
2  // criamos o nosso grafo orientado
3  SemanticGraph graph = this.getSemanticGraph(req);
4  SemgrexPattern sbj = SemgrexPattern.compile("{$} >/nsubj|
       nsubjpass/ {}=subj");
5  SemgrexMatcher matcher = sbj.matcher(graph);
6  String className = "";
7  String compound = "";
8
9  while (matcher.find()) {
10 for(IndexedWord word: graph.getChildrenWithReln(matcher.getNode("
       subj"),UniversalEnglishGrammaticalRelations.COMPOUND_MODIFIER){
11 compound += StringUtils.capitalize(getSingularForm(word)) +
       compound;
12 }
13 className = compound + StringUtils.capitalize(matcher.getNode("
       subj").word());
14 }
15 ExtractedClass retorno = new ExtractedClass("Actor", className);
16 return retorno;
17 }
18 }
```

Código 5.1 – Exemplo de código para percorrer e extrair dados do grafo de dependências.

As demais regras de extração são variações (algumas bem complexas) do apresentado em Código 5.1, com isso criamos uma classe ***RelaxPLNParser*** com uma série de métodos que realizam a extração de cada tipo de informação, e cada tipo de informação extraída (*Actor, Action, Environment* ...) segue suas próprias regras de extração. Todas as regras apresentadas a seguir foram desenvolvidas seguindo os conceitos explorados nesta seção.

5.3 Regras de Extração

O método utilizado para a criação das nossas regras foi o seguinte: 1) Com base no nosso *corpus* de requisitos, primeiro pegamos a sentença em LN, marcamos quais os elementos desejados, executamos o *parsing* na sentença, observamos o grafo gerado anotando quais os relacionamentos que oferecem as informações desejadas; 2) Depois disso escrevemos um teste unitário, onde a entrada é a sentença em LN, e o *Assert* é o nosso resultado esperado; 3) Na sequência a regra é codificadas em um método extrator, e testada usando o teste unitário. Realizamos refatorações no extrator sempre que a saída não é igual a esperada ou que algum novo relacionamento é necessário, realimentando o ciclo.

Utilizando da semântica dos operandos e operadores RELAX, os estereótipos e restrições do *profile* UML e as técnicas de PLN explorados acima, chegamos a um conjunto de regras de extração que definem como cada um dos elementos do modelo é extraído.

Separamos tais regras em duas categorias, as que podem ser extraídas sem a análise dos operadores da RELAX, as quais chamamos de "classes independentes"; e as classes que precisam da análise dos operadores para serem extraídas, as quais chamamos de "classes dependentes". Nessa última, a extração das classes ocorre simultaneamente à extração dos operadores.

5.3.1 Classes Independentes

Classes ***Environment***: a RELAX oferece a cláusula ENV, de onde podemos extraí-las, porém essas classes estão descritas em LN, uma etapa de PLN é adicionada aqui, buscando encontrar substantivos e palavras associadas a esse substantivo, ou em relacionamentos de pertinência, essas classes normalmente são quem possuem outras propriedades. Essas classes tendem a ter nomes curtos e em alguns casos compostos, como *Smartcar*, *Engine* ou *SmartcarEngine* por exemplo. As expressões *semgrex* utilizadas tratam duas possibilidades opostas de composição (por meio da *tag* "IN"):

```
{$}>>/nmod:poss/{tag:/NN.*/}=class|>>nmod({tag:/NN.*/}=class!>{tag:IN})
{$} >>nmod ({}=class >> {tag:IN})
```

Classes ***Monitor***: são oferecidas pela cláusula MON da RELAX, e o processo de PLN busca encontrar o objeto direto (quando existir) combinada com o principal substan-

tivo da sentença (sujeito), formulando nomes como *EngineSensor*, *EngineThermometer* por exemplo. As expressões *semgrex*:

```
{$} >>/nmod|.*obj/ ({}=obj ?>>/amod|compound/ {pos:/NN.*/}=objC)
{$} >>/.*subj/ {}=sbj ?>> compound{}=sbjC
```

Classes ***Relationship***: a RELAX oferece a cláusula REL para indicar relacionamentos entre classes *Environment* e *Monitor*, esses relacionamentos se dão por meio de classes associativas do tipo *Relationship*, (seguindo as definições e recomendações de modelagem do *profile* RelaxML) as quais são extraídas pela simples análise dessa cláusula. Conseguimos extrair essa classe, porém não conseguimos encontrar um padrão em nosso *corpus* de estudo para definirmos um nome para ela, normalmente ela recebe nomes com palavras que não estão contidas no requisito (implícito), tornando impraticável qualquer inferência válida, apesar disso, apresentamos como sugestão de nome, no campo "Informações Adicionais" o padrão mais comumente encontrado, que foi o nome da classe *Environment* mais o fragmento "Info", indicando que essa classe oferece informações sobre o ambiente.

Classes ***Actor***: essas classes são obtidas por meio de PLN puro atuando sobre o requisito como um todo (ignoramos a RELAX nessa etapa), basicamente procuramos um substantivo que seja o sujeito da sentença, isso em expressão *semgrex*:

```
{$} >/nsubj|nsubjpass/ {}=subj
```

Classes ***Action***: também utiliza PLN puro, procuramos por verbos que estejam relacionados ao sujeito, normalmente é o próprio nó raiz da sentença, portanto nesse caso procuramos apenas elementos que se relacionam diretamente com a raiz e que apresentem informações relevantes, como o objeto da sentença ou modificadores, isso em expressões *semgrex*:

```
{$} >/dobj/ {}=obj
{$} >/nmod/ {}=mod | >conj ({} >nmod {}=mod)
```

Classes ***Event***: alguns operadores da RELAX indicam a existência de um evento, o *profile* RelaxML recomenda que um evento deve ser representado por uma classe do tipo *Event*, seu nome é definido por PLN buscando uma ação dentro do operando "e" oferecido pela RELAX. Ações remetem a um verbo, que normalmente são a raiz do grafo, portanto nesse caso novamente apenas buscamos elementos que se relacionem diretamente com a raiz, buscando o objeto da sentença, em *semgrex*:

```
{$} >/dobj/ {}=obj
```

5.3.2 Classes Dependentes

A presença dos operadores é identificada por meio da análise da expressão RELAX do requisito, que indica todos os operadores do requisito, como "SHALL (BEFORE (e p1))" por exemplo. Fazemos uma análise simples para verificar a presença ou não de um operador específico. Em muitos casos as classes (operandos) são extraídas com base nos operadores e nas recomendações e regras OCL descritas no *profile* RelaxML. Além disso, o *profile* descreve os operadores RELAX como relacionamentos do tipo Dependência com um estereótipo associado. Abaixo descrevemos como os operadores são tratados quando encontrado em um requisito, e como ocorre a extração das classes dependentes (quando existirem).

Operador ***SHALL***: o *profile* indica que o operador *SHALL* deve sempre partir de uma classe do tipo *Actor* e ligar a outra do tipo *Action*, como a extração dessas classes (operandos) ocorre em uma etapa anterior, basta agora criar o relacionamento de dependência com seu devido esteriótipo.

Operador ***EVENTUALLY***: mesmas regras do *SHALL*, o que muda é sua semântica.

Operador ***MAY ... OR MAY***: esse operador quebra o requisito em duas ou mais sentenças com ações opcionais, elas compartilham o mesmo sujeito (*Actor*), mas apresentam ações (*Action*) separadas, portanto nesse caso, o requisito é quebrado em vários requisitos, um para cada cláusula "MAY" (damos um *split* em "OR MAY" e reformulamos os trechos quebrados para que todos contenham o sujeito), e dessa forma utilizamos em laço o *parsing* de extração de ações, portanto esse operador resulta em mais de uma classe do tipo *Action*. Depois da extração dessas classes, ocorre a criação dos relacionamentos, sempre partindo do único *Actor* do requisito, e ligando com uma classe do tipo *Action*, para a primeira *Action* extraída, o relacionamento é do tipo "MAY", para todas as demais *Actions* o relacionamento será do tipo "OR MAY".

Operador ***UNTIL*** e classes ***State***: o *profile* indica que esse operador sempre liga uma classe *Action* com uma classe do tipo *State*, portanto classes *State* são extraídas a partir disso. Análises em nosso *corpus* indicam que esse estado (*State*) normalmente está associado a ação a qual o requisito especifica, exemplo: "A plantadeira deve plantar até as sementes acabarem", onde plantar é a ação, e caixa de sementes vazia é o estado, porém por meio de PLN, obteríamos classes com nomes estranhos, como "SementesAcabarem", portanto não extraímos o nome de tais classes *State*, mas em contra partida, capturamos o estado esperado e apresentamos nas "Informações Adicionais" da classe extraída, essa informação pode ajudar o modelador a escolher um nome mais apropriado. Depois da extração da classe, basta criar a o relacionamento de dependência com o estereótipo *Until* entre a classe *Action* (extraída anteriormente) e a classe *State* para indicar o operador *UNTIL*.

Classes do tipo ***Timer*** são derivadas a partir de alguns operadores, como o **ope-**

rador *IN*, que indica um intervalo de tempo "t", as regras do *profile* recomendam que sempre que um intervalo de tempo é requerido, este deve ser representado por uma classe *Timer*, portanto extraímos uma classe *Timer*, e fixamos seu nome como "Clock", tendo em vista que o parâmetro "t" indica um intervalo fixo de tempo. Os **operadores *AS (EARLY, LATE) AS POSSIBLE*** são operadores temporais, porém aqui não temos um intervalo fixo definido, sua definição vem por meio de um conjunto *Fuzzy*. O *profile* recomenda que sempre que esses operadores estiverem presentes no requisito, uma classe *Timer* que vai representar o conjunto *Fuzzy* deve ser extraída, fixamos nomes para essas classes como "EarlySet" "LateSet", que indicam claramente seu papel no modelo.

Outro operador que atua sobre um conjunto *Fuzzy* temporal é ***AS CLOSE AS POSSIBLE TO f***, onde "f" (frequência) é um dado temporal, indicando um período que pode ser flexibilizado (diferente do *IN*). A semântica desse operador já define seu conjunto *Fuzzy*, mas novamente o nome da classe *Timer* extraída não pode ser inferido, nesse caso fixamos o nome como "TimeSet". Com a análise desses três operadores, identificamos a presença de classes do tipo *Timer*. O *profile* recomenda que esses operadores sempre ocorrem entre classes *Action* e *Timer*, portanto a ultima etapa é acrescentar ao modelo os relacionamentos de dependência que indicam qual o operador extraído.

Classes do tipo ***Quantifier***: são extraídas a partir dos **operadores *AS (MANY, FEW) AS POSSIBLE***, eles atuam sobre quantidades, porém essa quantidade pode ser flexibilizada, que é representada novamente em um conjunto *Fuzzy*. O *profile* recomenda que esses operadores devem atuar sobre classes do tipo *Quantifier* (que representam esse conjunto), aliado a isso a RELAX permite a especificação de "o que" está sendo quantificado, da seguinte forma: "AS MANY food packages AS POSSIBLE", portanto *FoodPackages* será o nome da nossa classe.

Quando isso ocorre, fazemos um tratamento para tornar essa informação capitalizada e sem espaços, para ser um nome válido para a classe. Porém esse tipo de construção é opcional, e nesses casos não temos a informação de "o que" está sendo quantificado, assim fixamos o nome como "ManySet" ou "FewSet" respectivamente.

Outro operador que atua sobre um quantificador é ***AS CLOSE AS POSSIBLE TO q***, onde "q" é um quantificador, o *profile* recomenda que esse operador também atue sobre classes do tipo *Quantifier*, porém aqui há o operando "q" (inexistente nos operadores MANY e FEW). O nome dessa classe novamente não pode ser inferido, portanto fixamos seu nome como "QuantitySet" e seu parâmetro "q" é passado para a classe extraída como "Informação Adicional".

Depois de a classe *Quantifier* extraída, basta adicionar a dependência com seu estereótipo entre a classe *Action* e *Quantifier*, para indicar a presença de um desses operadores no modelo.

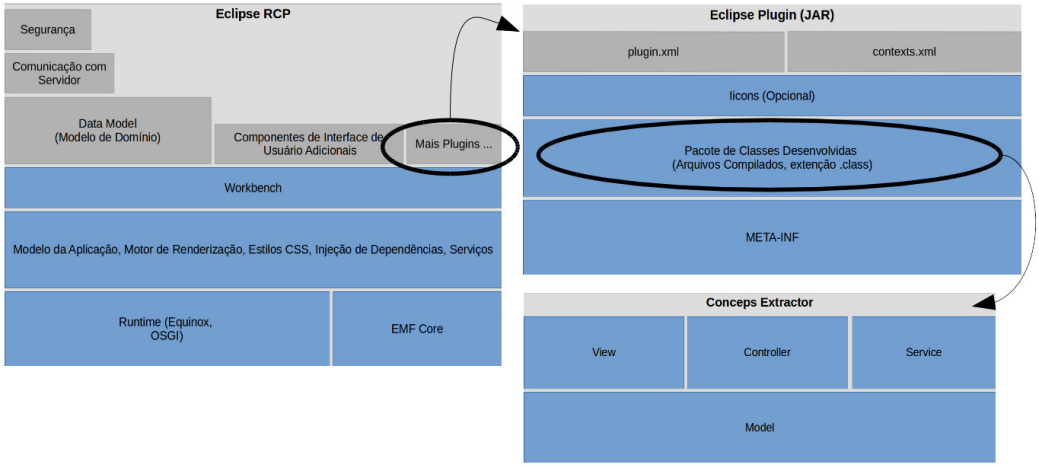
Operadores ***BEFORE*** e ***AFTER***: O *profile* indica que esses operadores sempre ocorrem entre uma classe *Action* e outra classe *Event*. A classe *Event* é extraída com

base no operando “e”, oferecido pela RELAX, portanto basta acrescentar a dependência entre as classes *Action* e *Event* para indicar a presença desses operadores;

5.4 Arquitetura da Solução

Nosso projeto é baseado na plataforma Eclipse, e a Figura 14 ilustra a arquitetura geral da nossa solução. O primeiro nível, eclipse RCP, é o nível mais alto da arquitetura, trata-se do nosso produto final que contém o editor de requisitos *Relax Editor* mais nosso extrator, estes são projetos distintos. No segundo nível, eclipse *plugin* , mostra a arquitetura interna de um *plugin* eclipse, a qual o nosso extrator teve de implementar. No terceiro nível apresentamos uma visão geral da nossa solução, interna ao *plugin* eclipse.

Figura 14 – Arquitetura geral da solução, em três níveis.



Fonte: O autor

Optamos por distribuir nossa ferramenta final como um produto eclipse RCP pois assim todos os recursos necessários para a edição de requisitos RELAX, oferecidos pela *Relax Editor*, mais a extração dos modelos conceituais já vem nativamente configuradas. Porém nada impede de distribuirmos, tanto o editor de requisitos quanto o extrator de forma independente, por meio de *plugins* que podem ser incorporados em instalações do eclipse IDE. Lembrando porém que o nosso extrator depende da saída gerada pelo *Relax Editor*.

5.4.1 Integração com a *Relax Editor*

Um dos objetivos do nosso trabalho é a integração do nosso extrator com o protótipo de ferramenta *Relax Editor*, originalmente desenvolvida por Moro (2015), onde ele propôs uma ferramenta de suporte à especificações de requisitos RELAX que imple-

menta toda a sua gramática, com as devidas análises e geração das expressões gramaticais RELAX.

No entanto, em nosso trabalho optamos por re-desenvolver a *Relax Editor* pelos seguintes motivos: a ferramenta original de Moro (2015) foi desenvolvida com uma versão antiga da plataforma Eclipse; A versão original não apresentava uma série de recursos desejados à edição de requisitos, como o *syntax high-light*, que indica erros de sintaxe em tempo de escrita, além do *auto-complete, hovering*, entre outros; a ferramenta apresentava alguns *bugs* que precisavam ser corrigidos. Portanto, deste ponto adiante, sempre que nos referirmos à *Relax Editor*, estamos nos referindo à nossa versão re-desenvolvida da ferramenta.

Para isso, utilizamos a ferramenta Xtext para processar as partes formais da RELAX, ela oferece recursos para a escrita e definição de linguagens, como vimos na subseção 3.2.1.

Algumas regras adicionais foram acrescentadas em relação à ferramenta original de Moro (2015) e à gramática da RELAX, são elas: a palavra reservada **req** ou **REQ**, que indica o início de um requisito, e um **identificador** único **para os requisitos**; os blocos ENV, MON, REL e DEP foram formalmente definidos em nossa gramática, onde tivemos de definir um **identificador** único **para cada elemento** declarado dentro **dos blocos ENV ou MON**, a fim de possibilitar seu relacionamento, utilizando o bloco REL. Os blocos ENV, MON, REL e DEP são apresentados no artigo de Whittle et al. (2010), porém lá são tratados apenas como blocos de informações adicionais. Outras adições da gramática dizem respeito a possibilidade de expressar uma lista de requisitos, utilizando o caractere ";" para delimitar o fim de um requisito, e o caractere escape "\" que usamos para "escapar" operadores, como a palavra "in" e "to" do exemplo Código 5.2.

```
1 REQ prenat01: The smartphone SHALL send a help message \to
     another authorized user when the pregnant woman is \in a
     hospital;
2 ENV{currentPosition: Current position of the pregnant woman; }
3 MON{gps: The pregnant woman's smartphone GPS;}
4 REL{gps - currentPosition;}
```

Código 5.2 – Requisito RELAX escrito com a *Relax Editor*.

O exemplo Código 5.2 mostra como um requisito RELAX deve ser escrito usando a ferramenta *Relax Editor*, observe que os blocos ENV, MON, REL e DEP são opcionais, e devem aparecer depois do término do requisito ";". A saída produzida para esse requisito é apresentada em Código 5.3.

```
1 ReqId:=prenat01
2 exp:=SHALL( p1);
3 p1:=The smartphone send a help message \to another authorized
     user when the pregnant woman is \in a hospital
```

```
4
5 ENV:=currentPosition: Current position of the pregnant woman
6 MON:=gps: The pregnant woman's smartphone GPS
7 REL:=mon=gps -> env=currentPosition
```

Código 5.3 – Expressão gerada pela *Relax Editor*.

Nossa versão da *Relax Editor* oferece as análises léxicas, sintáticas, e algumas análises semânticas, como a verificação de identificadores únicos por exemplo, além de oferecer o recurso de geração das expressões gramaticais RELAX, as quais utilizamos na extração dos conceitos. A gramática da RELAX que desenvolvemos nessa reescrita, foi codificada na *grammar-language* oferecida pela Xtext (ver subseção 3.2.1) e é apresentada no Apêndice A.

5.4.2 *Plugin Concepts Extractor*

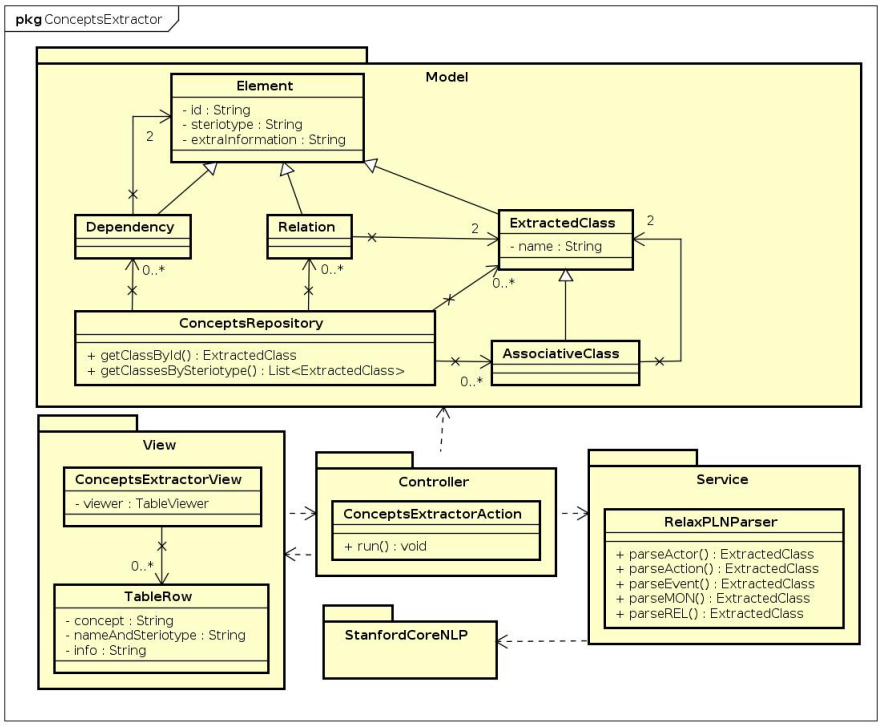
O principal objetivo do nosso trabalho é realizar a extração de modelos conceituais partindo de requisitos RELAX. Portanto a utilização da *Relax Editor* oferece um passo intermediário nesse processo. Codificamos e desenvolvemos nossas regras de extração em um *plugin* eclipse, que partindo dos requisitos e expressões RELAX escritos com a *Relax Editor*, gera um modelo conceitual de classes, exibindo os conceitos e relacionamentos em uma lista, de forma textual. Chamamos nosso *plugin* de ***Concepts Extractor***.

A Figura 15 apresenta a arquitetura interna do nosso *plugin*, as duas principais camadas da são a camada *Model*, que é onde está nosso modelo de domínio, nela estão os elementos que usamos para representar o diagrama conceitual de classes extraído, e a camada *Service* é onde nossas regras de PLN estão codificadas. As demais camadas implementam partes da arquitetura de um *plugin* eclipse.

A plataforma Eclipse tem seu *workbench* dividido em seu nível mais alto entre *views* e *editors* (ver seção 3.1). *editors* como o nome sugere são os editores, sejam de código ou de diagramas, a *Relax Editor* estendeu um *editor*. Já as *views* são as outras janelas do Eclipse, como a *view Project Browser*, para navegar na estrutura de diretórios por exemplo.

Nossa solução, o *plugin Concepts Extractor*, cria uma nova *view* para o eclipse, ele processa o requisito RELAX ativo no *editor*, e lista as extrações em uma *view* (normalmente apresentada ao lado do *Console*, mas isso é configurável). Por fim, o *Concepts Extractor* foi adicionado à lista de *plugins* do produto eclipse RCP que criamos, e portanto já vem nativamente instalado à *Relax Editor*. Caso o *plugin Concepts Extractor* não estiver visível, basta ir até o menu *Window – Show View – Other* e digitar "*concepts extractor*" no campo de busca, selecioná-lo e clicar em "Ok" para que se torne visível.

Figura 15 – Modelo conceitual e arquitetura da nossa solução *Concepts Extractor*

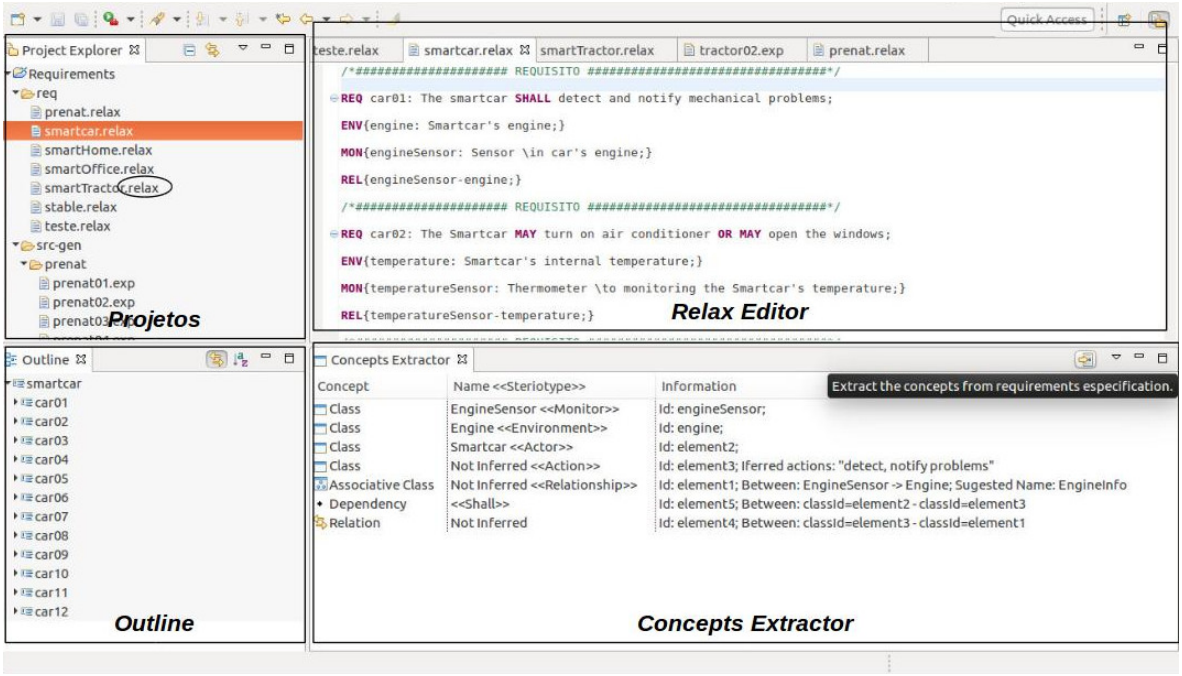


Fonte: O autor

5.5 Visão Geral do Protótipo

Para finalizar, a Figura 16 apresenta a visão geral do nosso protótipo de ferramenta, a *Relax Editor*, com o *plugin Concepts Extractor* ativo. Começando pela esquerda e no topo, temos a *view* de projetos, onde podemos navegar na estrutura de diretórios do(s) nosso(s) projeto(s). Atente para o detalhe na extensão dos arquivos, para que a *Relax Editor* dê suporte à linguagem RELAX, os arquivos devem estar com a extenção ".relax". Logo abaixo temos o *outline*, que lista todos os elementos presentes no editor em uma estrutura de árvore, essa estrutura representa o nosso AST, e é possível navegar pelo *editor* por meio dessa árvore. À direita e no topo é apresentada nossa nova versão do editor de requisitos RELAX, re-desenvolvido a partir do trabalho de Moro (2015), perceba que seus operadores estão em uma cor diferente, indicando a análise léxica em tempo de edição de código, além disso, o protótipo realiza a análise sintática, exibindo mensagens de erro sempre que a sitáxe não for respeitada. E por fim, abaixo do editor de requisitos, é apresentado o núcleo do nosso trabalho, o *plugin Concepts Extractor*, listando um modelo conceitual de forma textual.

Figura 16 – O protótipo da ferramenta ***Relax Editor*** com nosso *plugin* ***Concepts Extractor*** ativo.



Fonte: O autor

5.6 Lições do Capítulo

Neste capítulo apresentamos as ferramentas e artefatos utilizados na construção do nosso protótipo, que pode ser dividia em dois, o ***Relax Editor***, que é um editor funcional de requisitos RELAX e gerador de expressões originalmente proposto por Moro (2015) e re-desenvolvido por nós, e o ***Concepts Extractor***, que é onde está o nosso trabalho, atuando como um complemento da *Relax Editor* na extração dos modelos conceituais.

Utilizamos dos recursos da plataforma Eclipse, que se mostrou bastante útil para o desenvolvimento de ferramentas voltadas ao desenvolvedor, destacamos a Xtext, que torna simples o processo de desenvolvimento de uma nova linguagem, pontuamos a facilidade de integração com eclipse e sua extensividade por meio de *plugins*.

Por fim, utilizamos de recursos do *Stanford CoreNPL* para as nossas regras de extração implementadas no *plugin Concepts Extractor*.

6 VALIDAÇÃO DO PROTÓTIPO

Nesse capítulo descrevemos como ocorreu a validação da nossa ferramenta de extração de conceitos, na seção 6.1, apresentamos e justificamos nosso método de avaliação. A seção 6.2 explica quais as medidas usadas, como coletar os dados necessários e como conduzir a avaliação. A seção 6.3 apresenta os resultados do nosso processo de avaliação. Na seção 6.4 discutimos esses resultados, comparando nossos resultados com os de alguns trabalhos relacionados e por fim na seção 6.5 apresentamos as lições aprendidas nesse capítulo.

6.1 Avaliação

Com o objetivo de validar nossa solução, desenvolvemos uma avaliação com base nos trabalhos de Elbendak, Vickers e Rossiter (2011), Sharma, Srivastava e Biswas (2015), Sagar e Abirami (2014) e Afreen, Bajwa e Bordbar (2011), apresentados no Capítulo 4, esses trabalhos apresentam o mesmo conjunto de medidas que servem para avaliar ferramentas extratoras, como a nossa.

Em um primeiro momento, realizamos a avaliação com base nessas medidas, e depois comparamos os nossos resultados com os resultados dos trabalhos citados, caracterizando duas formas distintas de avaliação.

As medidas utilizadas refletem a performance da nossa ferramenta na atividade de extrair modelos conceituais de requisitos no que diz respeito a precisão e completude. O critério utilizado nesse tipo de avaliação é: Relacionar o quão próximo o modelo extraído está de um modelo "certo".

Sabemos que não existe um único modelo correto para qualquer software sob desenvolvimento, os modelos variam de acordo com a área de domínio e experiência do modelador, diferentes analistas humanos normalmente produzem diferentes modelos, e esse modelos não podem ser categorizados como estritamente "corretos" ou "incorretos", mas podem ser categorizados em modelos "bons" ou "ruins", de acordo com as classes e relacionamentos presentes (ELBENDAK; VICKERS; ROSSITER, 2011). Em nosso trabalho assumimos os modelos do cenário *Smartcar* como "**modelo referência**" para a nossa avaliação, tais modelos podem ser vistos no Apêndice C. Os requisitos que os analistas usaram para gerar esses modelos podem ser vistos no Apêndice B e ambos foram desenvolvidos e revisados pelos membros do grupo LESA da UNIPAMPA.

Para garantir que esses modelos estão o mais correto possível, eles passaram por um processo de desenvolvimento e revisão colaborativo, onde cada modelo foi revisado por quatro pessoas, sendo três alunos do curso de Engenharia de Software, entre o quinto e oitavo semestre, e um professor da UNIPAMPA. O modelo só foi considerado "pronto" e "correto", quando todos os envolvidos concordavam com ele. Esse cenário apresenta um total de 12 modelos, um para cada requisito.

6.2 Medidas

Usamos duas medidas que vieram originalmente da área de recuperação da informação (*information retrieval*) são elas: *precicion*, *recall*, essas são medidas básicas para avaliação de qualquer estratégia de recuperação de dados. Essas medidas idealmente devem estar o mais próximos de 100% quanto possível. Os trabalhos de Elbendak, Vickers e Rossiter (2011), Sharma, Srivastava e Biswas (2015) e Sagar e Abirami (2014) utilizaram uma terceira medida, chamada ***Over-specification***, que reflete a quantidade de dados extra que a ferramenta produz. Nessa medida, de acordo com Elbendak, Vickers e Rossiter (2011), um elemento é considerado extra se está correto mas não está presente no modelo referência. Em nossa avaliação não consideramos *over-specification*, por considerarmos nosso modelo referência correto e completo.

Recall reflete o quão completo é o resultado obtido, as informações corretas e relevantes retornadas pelo extrator são comparadas com o modelo referência, o percentual *recall* indica a habilidade de automaticamente gerar os elementos presentes no modelo referência. *Recall* é definido com a seguinte fórmula:

$$Recall = N_{correct} \ / \ (N_{correct} + N_{missing})$$

Onde $N_{correct}$ diz respeito ao número de extrações corretas feitas pelo extrator, e $N_{missing}$ é o número de elementos presentes no modelo referência que não foram extraídas pelo extrator.

Precision reflete a acurácia do modelo, ou seja, o quanto de informação extraída está correta em relação ao modelo referência. É definido com a seguinte fórmula:

$$Precision = N_{correct} \ / \ (N_{correct} + N_{incorrect})$$

Onde $N_{incorrect}$ diz respeito ao número de extrações incorretas feitas pelo extrator.

Dividimos nossa avaliação em duas, seguindo critérios diferentes para os valores de $N_{correct}$, $N_{incorrect}$ e $N_{missing}$. Algo semelhante ocorreu no trabalho de Sagar e Abirami (2014), onde o autor separa a avaliação em conceitos implícitos e explícitos. No nosso caso, separamos a avaliação das classes e relacionamentos quanto aos seus **valores** (nome, estereótipo) e quanto ao seu **significado** (semântica), pois em muitos casos nosso extrator não define o nome da classe, mas define seu estereótipo, informações adicionais e relacionamentos. Assim conseguimos avaliar o quão corretas estão os elementos que extraímos, considerando outras informações além do nome estereótipo.

Um método manual foi definido para produzir os valores $N_{correct}$, $N_{missing}$, $N_{incorrect}$ e N_{extra}. Vejamos:

$\boldsymbol{N_{correct}}$ para avaliação dos **valores**: Classes (e classes associativas) são consideradas corretas quando seu nome e seu esteriótipo são idênticos aos do modelo referência.

$N_{correct}$ para avaliação dos **significados**: Classes (e classes associativas) são consideradas corretas quando seu nome e seu esteriótipo são idênticos aos do modelo referência ou quando o nome é diferente, mas claramente indicam a mesma entidade. Usamos os requisitos e nosso próprio julgamento para decidir se trata-se do mesmo elemento. Para classes as quais não decidimos um nome (*Not inferred*), elas são consideradas corretas quando o esteriótipo é o mesmo do modelo referência, e realizamos uma análise utilizando das informações adicionais e nosso julgamento para decidir se trata-se da mesma classe.

Em **relacionamentos** (seja dependência ou associação) consideramos corretos apenas os que ocorrem entre classes consideradas corretas e que estejam com o estereótipo correto (quando existir).

$N_{incorrect}$ para avaliação dos **valores**: Para qualquer classe (associativa ou não) que tenha um nome ou estereótipo diferente do apresentado no modelo referência, contamos como incorreta. Para qualquer relacionamento (seja associação ou dependência) que esteja entre dois elementos diferentes do modelo referência, contamos como incorreto. Para qualquer dependência que apresentar um estereótipo diferente do modelo de referência, contamos como incorreto. Para qualquer classe que o extrator não consegue definir um nome, contamos como incorreto. Para qualquer relacionamento que ocorre com pelo menos uma classe incorreta, contamos como incorreto.

$N_{incorrect}$ para avaliação dos **significados**: Para qualquer classe (associativa ou não) que tenha um estereótipo diferente do apresentado no modelo referência, contamos como incorreta. Para qualquer relacionamento (seja associação ou dependência) que esteja entre dois elementos diferentes do modelo referência, contamos como incorreto. Para qualquer dependência que apresentar um estereótipo diferente do modelo de referência, contamos como incorreto.

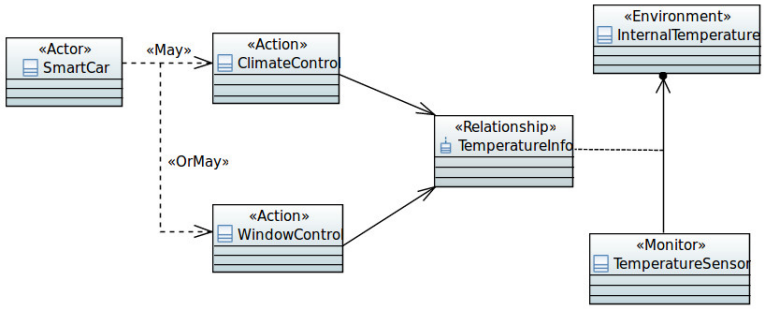
$N_{missing}$ para avaliação dos **valores** e dos **significados**: Para qualquer elemento presente no modelo referência e não identificado pelo nosso extrator, contamos como faltante (*missing*).

Vejamos um exemplo de como ocorreu a contagem do requisito car02, em Código 6.1 apresentamos o requisito, na Figura 17 apresentamos o modelo referência do requisito, e por fim na Figura 18 apresentamos a saída gerada pelo nosso *plugin Concepts Extractor*, o qual apresenta um modelo conceitual de classe de forma textual.

```
1 REQ car02: The Smartcar MAY turn on air conditioner OR MAY open
     the windows;
2
3 ENV{temperature: Smartcar's internal temperature;}
4 MON{temperatureSensor: Thermometer \to monitoring the Smartcar's
     temperature;}
5 REL{temperatureSensor-temperature;}
```

Código 6.1 – Requisito RELAX car02.

Figura 17 – Modelo referência do requisito car02.



Fonte: O autor

Figura 18 – Saída do *plugin Concepts Extractor* para o requisito car02.

Tasks | Concepts Extractor

Concept	Name <<Steriotype>>	Information
Class	TemperatureThermometer <<Monitor>>	Id: temperatureSensor;
Class	Temperature <<Environment>>	Id: temperature;
Class	Smartcar <<Actor>>	Id: element2;
Class	Not Inferred <<Action>>	Id: element3; Iferred actions: "turn on air conditioner"
Class	Not Inferred <<Action>>	Id: element5; Iferred actions: "open windows"
Associative Class	Not Inferred <<Relationship>>	Id: element1; Between: TemperatureThermometer —> Temperature; Sugested Name: TemperatureInfo
Dependency	<<May>>	Id: element4; Between: id=element2 —> id=element3
Dependency	<<OrMay>>	Id: element6; Between: id=element4 —> id=element5
Relation	Not Inferred	Id: element7; Between: id=element3 —> id=element1
Relation	Not Inferred	Id: element8; Between: id=element5 —> id=element1

Fonte: O autor

Agora basta fazer a contagem dos elementos, tanto para os valores quando para os significados dos elementos extraídos. Por exemplo, a primeira classe apresentada na Figura 18, apresenta o nome "*TemperatureThermometer*" e o estereótipo "*Monitor*". Na **avaliação por valor**, essa classe conta como incorreta, pois não há no modelo referência (Figura 17) nenhuma classe com esse nome. Porém quando fazemos a **avaliação por significado**, considerando a semântica da classe e não seu nome, fica claro que essa classe é equivalente à classe "*TemperatureSensor*", presente no modelo referência apresentado na Figura 17, portanto conta como correta nesse caso.

Esse processo de contagem foi repetido para todos os elementos de todos os requisitos (os modelos de referência estão no Apêndice C, e os requisitos no Apêndice B), e os resultados serão apresentados na próxima seção.

6.3 Resultados

Produzimos os valores de $N_{correct}$, $N_{missing}$ e $N_{incorrect}$ para os doze modelos gerados pelo nosso extrator.

Avaliação de Valores: a Tabela 5 mostra os resultados. Nessa tabela catego-

rizamos os dados coletados em dois conjuntos diferentes, um relacionado as classes do modelo extraído, e outro aos relacionamentos (como associações e dependências), e por fim mostramos a soma desses dois conjuntos.

Tabela 5 – Tabela das medidas intermediárias por requisito para os **Valores**.

	car01	car02	car03	car04	car05	car05	car07	car08	car09	car10	car11	car12
Classes												
$N_{correct}$	3	1	2	1	3	4	3	3	4	4	3	3
$N_{incorrect}$	2	5	3	4	3	2	4	4	3	3	4	4
$N_{missing}$	2	5	3	4	3	2	4	4	3	3	4	4
Relacionamentos												
$N_{correct}$	0	0	0	0	0	0	0	0	0	0	0	0
$N_{incorrect}$	2	4	2	3	3	3	4	4	4	5	4	4
$N_{missing}$	2	4	2	3	3	3	4	4	4	5	4	4
Totais												
$N_{correct}$	3	1	2	1	3	4	3	3	4	4	3	3
$N_{incorrect}$	4	9	5	7	6	5	8	8	7	8	8	8
$N_{missing}$	4	9	5	7	6	5	8	8	7	8	8	8

Com base nas nos dados apresentados na Tabela 5, podemos calcular nossos resultados. A Tabela 6 apresenta os percentuais (arredondados) das nossas duas medidas para cada um dos requisitos para avaliação dos valores.

Tabela 6 – Resultados por requisito (em percentual aproximado) para os **valores**.

	car01	car02	car03	car04	car05	car05	car07	car08	car09	car10	car11	car12
Classes												
Recall	60	16,67	40	20	50	66,67	42,85	50	66,67	57,14	42,85	42,85
Precision	60	16,67	40	20	50	66,67	42,85	50	66,67	57,14	42,85	42,85
Relacionamentos												
Recall	0	0	0	0	0	0	0	0	0	0	0	0
Precision	0	0	0	0	0	0	0	0	0	0	0	0
Totais												
Recall	42,85	10	28,57	12,5	33,33	44,44	27,27	30	40	33,33	27,27	27,27
Precision	42,85	10	28,57	12,5	33,33	44,44	27,27	30	40	33,33	27,27	27,27

Partindo da Tabela 6, podemos perceber que *precision* e *recall* ficaram com os mesmos valores. Isso se deve ao fato de que nosso extrator sempre extraiu o mesmo número de classes das presentes no modelo referência. Primeiro contamos as classes incorretas, depois as *missing*, nesse caso em específico, uma classe incorreta implica também numa classe *missing*, pois se uma classe está incorreta, a presença de uma correta no modelo extraído continua faltando, o que conta como *missing*.

Devido a isso, ambas as medidas ficaram com os mesmos valores, recuperamos um percentual baixo de classes consideradas corretas, com *recall* médio de $\simeq$43,31%. A extração de classes ficou com uma precisão baixa, $\simeq$43,31%, e a extração de relacionamentos, atingimos 0% de precisão, isso se deve a definição de que um relacionamento só é

considerada correto, se ocorrer entre duas classes corretas, e como a precisão das classes está baixa, isso afetou negativamente essa medida.

Os resultados do *Concepts Extractor* para a avaliação dos **valores** extraídos é obtido por meio de média aritmética simples:

- ***Recall:*** ≃29,74%;
- ***Precision:*** ≃29,74%;

Como podemos ver, obtivemos os mesmos valores para *precision* e *recall*. Esses valores podem ser considerados baixos, porém salientamos que nossos critérios de avaliação por valor, tornaram essa avaliação um tanto rígida, o que invalidou as extrações de todos os relacionamentos, contribuindo para esses baixos resultados. Além disso, em muitas classes nós optamos por não definir o nome, mas a extração (sem nome) tem valor, pois reconhecemos a existência de uma entidade, conseguimos atribuir estereótipos, relacionamentos e informações adicionais a ela, por essa razão uma avaliação considerando os significados (semântica) das classes e não apenas seu nome é importante em nosso trabalho.

Avaliação de Significados: a Tabela 7 mostra os resultados. Nessa tabela categorizamos os dados coletados em dois conjuntos diferentes, um relacionado as classes do modelo extraído, e outro aos relacionamentos (como associações e dependências), e por fim mostramos a soma desses dois conjuntos.

Tabela 7 – Tabela das medidas intermediárias por requisito para os **Significados**.

	car01	car02	car03	car04	car05	car05	car07	car08	car09	car10	car11	car12
Classes												
$N_{correct}$	5	6	5	5	6	6	7	7	7	7	7	7
$N_{incorrect}$	0	0	0	0	0	0	0	0	0	0	0	0
$N_{missing}$	0	0	0	0	0	0	0	0	0	0	0	0
Relacionamentos												
$N_{correct}$	2	4	2	3	3	3	4	4	4	5	4	4
$N_{incorrect}$	0	0	0	0	0	0	0	0	0	0	0	0
$N_{missing}$	0	0	0	0	0	0	0	0	0	0	0	0
Totais												
$N_{correct}$	7	10	7	8	9	9	11	11	11	12	11	11
$N_{incorrect}$	0	0	0	0	0	0	0	0	0	0	0	0
$N_{missing}$	0	0	0	0	0	0	0	0	0	0	0	0

Com base nas nos dados apresentados na Tabela 7, calculamos nossos resultados, apresentados na Tabela 8, com os percentuais (arredondados) das nossas duas medidas para cada um dos requisitos para avaliação dos significados.

A performance do *Concepts Extractor* quanto aos **significados** ou semântica das classes, é calculada por meio da média aritmética simples, os valores obtidos são:

Tabela 8 – Resultados por requisito (em percentual aproximado) para os **Significados**.

	car01	car02	car03	car04	car05	car05	car07	car08	car09	car10	car11	car12
Classes												
Recall	100	100	100	100	100	100	100	100	100	100	100	100
Precision	100	100	100	100	100	100	100	100	100	100	100	100
Relacionamentos												
Recall	100	100	100	100	100	100	100	100	100	100	100	100
Precision	100	100	100	100	100	100	100	100	100	100	100	100
Totais												
Recall	100	100	100	100	100	100	100	100	100	100	100	100
Precision	100	100	100	100	100	100	100	100	100	100	100	100

- ***Recall :*** 100%;
- ***Precision :*** 100%;

Quando consideramos a semântica das classes, nossos resultados são os ideais para as duas grandezas.

6.4 Discussões

Como nosso extrator não define o nome para algumas classes, essa forma de avaliação, utilizando dois critérios diferentes é necessária para avaliar o que nos propomos a extrair.

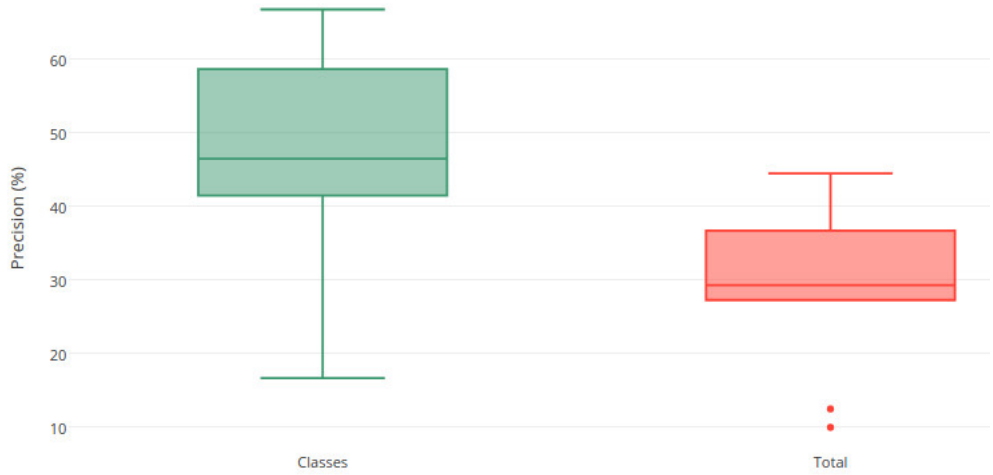
Na avaliação dos **valores**, apresentados na Tabela 5, o *precision* e *recall* na extração de classes está baixa (46,31%). Em muitas classes nosso extrator não define seu nome, e com nossa definição das medidas, contamos como um $N_{incorrect}$. Outro fator que influencia muito nesses números é que para uma classe ser considerada $N_{correct}$, o nome deve ser **idêntico** ao apresentado no modelo referência, e tivemos muitos casos em que nosso extrator definiu nomes com a mesma semântica, mas não idênticos, como por exemplo: "SmartcarGPS" ao invés de "GPS", ou "TemperatureThermometer" ao invés de "TemperatureSensor", e apesar de claramente essas classes terem a mesma semântica, são contadas como um $N_{incorrect}$.

A baixa precisão na extração de classes afeta negativamente a extração dos relacionamentos, pois seguindo nossas definições das medidas, um relacionamento só pode ser considerado correto, se ocorrer entre duas classes corretas. Devido a isso, todos os relacionamentos contaram como $N_{incorrect}$ (ver Tabela 5). Parte disso é devido a estratégia de modelagem do *profil* RelaxML, onde todos os elementos do modelo se relacionam com a classe *Action*, e como não definimos um nome para essa classe, a mesma contou como incorreta, portanto tornou todos os relacionamentos do modelo incorretos. Essas definições rígidas das medidas quando consideramos os valores (nome e estereótipo), aliadas a limitações do extrator, fizeram o *recall* e *precision* cair bastante, para **29,74%**.

O gráfico da Figura 19, apresenta a precisão (*precision*) da extração de classes e da precisão total do extrator considerando os valores das classes. Podemos observar que os resultados da extração de classes apresentam uma dispersão maior em relação ao Total, essa dispersão indica uma variação maior dos percentuais obtidos na extração de classes. Além disso, observamos que o gráfico do Total esta abaixo do de Classes, o que indica uma performance geral inferior. Sabemos que o que causou essa queda de performance no gráfico do Total foram os dados das extrações de relacionamentos, que apresentaram precisão zero nesse caso.

A mediana de ambos gráficos indicam que os dados são positivamente assimétricos, porém pontuamos a presença de dois *outliers* no Total, indicando casos em que a precisão da extração de classes ficou anormalmente baixa. Esses *outliers* são dos requisitos car02 (com 10%) e car04 (com 12,5%), O car02 apresentou um resultado anormalmente baixo, pois conta com o operador *MAY ... OR MAY*, que adiciona mais classes do tipo *Action*, as quais não possuem nome e são consideradas incorretas, além disso, por apresentar mais classes, apresenta também mais relacionamentos, que também estão incorretos quando seguimos essas definições de medidas. Já o requisito car04 apresentou esse resultado anormalmente baixo devido às classes *Action, State e Relationship* que não definimos o nome e contam portanto como incorretas, apenas uma das cinco classes presentes no modelo foi considerada correta, isso diminuiu para zero o número de relacionamentos corretos.

Figura 19 – Gráfico da precisão da extração de classes e da precisão total.



Fonte: O autor

Na avaliação dos **significados**, obtivemos os resultados considerados ideais para as duas grandezas. Nesse caso, consideramos as informações que extraímos para realizar a avaliação, como as informações adicionais por exemplo. Nas classes que não definimos

os nomes, consideramos seu esteriótipo, relacionamentos e informações adicionais para certificar que se trata da mesma classe, e nesse caso obtivemos 100% de precisão na extração de classes, isso afetou positivamente a extração dos relacionamentos, onde também atingimos 100% de precisão.

Quando consideramos a semântica das classes, e não seus nomes, nosso extrator apresenta resultados até melhores que os presentes no modelo referência, como no caso do requisito car11, onde o modelo referência apresenta uma classe do tipo *Quantifier* com o nome "*ManySet*", e nosso extrator definiu o nome como "*AlternativeRoutesQuantity*", indicando claramente do que se trata essa classe. Outro caso como esse ocorre no requisito car12, onde o extrator definiu o nome de uma classe como "*RefuelingPointQuantity*" ao invés de "*FewSet*" apresentado no modelo referência.

Analisando a Tabela 7, podemos perceber também que a extração de relacionamentos, é 100% precisa, todos os relacionamentos extraídos estão de acordo com o modelo de referência, isso em parte se deve à definição da própria linguagem RELAX, que apresenta o bloco **REL**, onde estão explicitamente definidos alguns dos relacionamentos do modelo, outro ponto que nos auxiliou nesse sentido foi o *profile* RelaxML, que apresenta uma série de restrições, as quais utilizamos ao nosso favor no momento da extração.

Não há ocorrências de $N_{missing}$ na avaliação de **significados**. Isso indica que nosso extrator encontrou todos os elementos presentes no modelo referência. Isso é de fato facilmente verificável, pois os modelos gerados pelo nosso extrator contém exatamente as mesmas classes (semanticamente falando) e relacionamentos dos apresentados no modelo referência, as diferenças estão apenas nos nomes das classes. Isso por sua vez aumentou o *recall* para 100% (pois não há elementos faltantes no modelo, $N_{missing}$).

Em comparação com alguns trabalhos relacionados, apresentados na Tabela 9, percebemos que nossa abordagem (duas últimas linhas), junto com Sharma, Srivastava e Biswas (2015), atingimos 100% em *Recall*. Quando consideramos os **valores** das classes, nossa ferramenta apresenta o pior desempenho na grandeza *precision*, isso muito se deve a rigidez com a qual avaliamos nossas extrações. Nesse caso, cabe salientar que o trabalho de (ELBENDAK; VICKERS; ROSSITER, 2011) avalia as classes de acordo com sua semântica, e não com seu nome.

Porém se considerarmos a avaliação dos **significados** das classes, obtivemos o melhor resultado de todos os trabalhos na grandeza *precision*. Porém cabe salientar que nosso extrator não define um nome para todas as classes, extraindo uma classe correta, mas incompleta.

Por fim cabe comentar que os dados apresentados na Tabela 9 foram gerados com conjuntos distintos de requisitos. Em nosso trabalho avaliamos um conjunto de requisitos que nós criamos, ao passo que os trabalhos comparados, utilizaram seus próprios conjuntos de requisitos.

Tabela 9 – Comparação com trabalhos relacionados.

Trabalho	***Recall***	***Precision***
(ELBENDAK; VICKERS; ROSSITER, 2011)	90%	85%
(SHARMA; SRIVASTAVA; BISWAS, 2015)	100%	≃91,6%
(SAGAR; ABIRAMI, 2014)	74,11%	≃89,17%
(AFREEN; BAJWA; BORDBAR, 2011)	83,82%	91,01%
Concepts Extractor **Valores**	≃ 29,74%	≃ 29,74%
Concepts Extractor **Significados**	100%	100%

6.5 Lições Aprendidas

A utilização de métodos de avaliação já consolidados proporcionam maior segurança em termos teóricos, além de possibilitar uma base para comparações. Ficou claro a importância de um método da avaliação apropriado com o objetivo do objeto avaliado.

Nosso protótipo apresentou um resultado geral bom, mas por meio desse método de avaliação, podemos comparar nossos resultados com os de outros trabalhos, e isso indicou uma boa performance em alguns aspectos, porém o mais importante foi a indicação de pontos onde podemos melhorar, como na extração de classes.

7 CONCLUSÕES

O principal objetivo do nosso trabalho é a criação de uma ferramenta capaz de extrair modelos conceituais a partir de requisitos textuais para SAs expressos na linguagem RELAX. Essa ferramenta vai auxiliar engenheiros de software na crítica tarefa de criação de modelos, e visa automatizar essa atividade, o que agiliza esse processo.

Dos objetivos específicos, consideramos que todos eles foram atingidos: primeiro, a exploração do formalismo da RELAX foi alcançado pois tiramos vantagem de suas definições formais e sua semântica nas nossas regras de extração, isso com o auxílio da ferramenta *Relax Editor*; segundo, desenvolvemos um conjunto de regras de extração para os elementos de um modelo conceitual, tendo como base a linguagem RELAX, o *profile* UML RelaxML, e principalmente técnicas de PLN, onde utilizamos os recursos oferecidos pelo *Stanford CoreNLP*; terceiro, incorporamos nosso extrator *Concepts Extractor* à ferramenta *Relax Editor* por meio de um *plugin* que desenvolvemos, onde estão codificadas nossas regras de extração. Mas para isso, optamos por re-desenvolver a *Relax Editor*, o que possibilitou oferecer um editor de requisitos mais completo que a versão inicial de Moro (2015). Por fim, empacotamos a nossa versão da *Relax Editor* juntamente com nosso *plugin Concepts Extractor* em um produto eclipse, o que possibilita tanto a edição e escrita de requisitos RELAX quanto a extração de modelos conceituais, atingindo portando o nosso objetivo geral de criar uma ferramenta que realiza extração de modelos conceituais a partir de requisitos RELAX.

O projeto que desenvolvemos combina elementos de diferentes áreas do conhecimento, o que o tornou desafiador. Um desafio o qual podemos pontuar, ocorreu no desenvolvimento dos analisadores e ferramental de suporte para a RELAX, pois a Xtext não suporta gramáticas recursivas à esquerda, fato que ocorre na RELAX. Outro ponto foi definir como ou onde ocorreria a combinação das técnicas de processamento formal com PLN, em alguns casos, optamos por utilizar apenas PLN (como na extração do ator e da ação, pois apresentam um resultado melhor). Porém o maior desafio foi desenvolver nossas regras de extração, pois em nossos trabalhos relacionados, não encontramos nenhuma abordagem que utiliza *parsing* de dependências, tivemos de estudar e entender a semântica dos relacionamentos e encontrar padrões em um *corpus* limitado de requisitos para então codificá-los em regras que retirem informações úteis dos requisitos.

Nosso protótipo aceita a gramática da RELAX em sua totalidade, permitindo sentenças complexas e grandes, porém para realizarmos a extração, tivemos de controlar essa natureza livre, permitindo o processamento de apenas um requisito por vez, e apenas uma ocorrência de cada operador da RELAX, caracterizando uma limitação.

Outro ponto fraco do nosso trabalho está na limitação do nosso *corpus* de requisitos e modelos onde estudamos para encontrar os padrões que geraram nossas regras, contávamos com um número relativamente pequeno de requisitos, e muitos deles tiveram de ser desenvolvidos por nós.

Apesar dos desafios e limitações citados, o protótipo apresentou um bom resultado em nossa avaliação, com *recall* em 100%, *over-specification* 0% (perfeito para essa medida) e uma precisão (*precision*) em 65%. Em comparação com trabalhos relacionados, obtivemos o melhor resultado em *over-specification*, porém cabe salientar que nossa avaliação ocorreu apenas sobre um cenário, explorando apenas um domínio.

7.1 Contribuições

Em nosso trabalho desenvolvemos o protótipo funcional de uma ferramenta que partindo de uma especificação de requisitos RELAX, extrai um modelo conceitual, automatizando parte do processo de desenvolvimento.

Essa ferramenta auxilia analistas de requisitos em projetos de SAs, oferecendo suporte ferramental nas tarefas de especificação de requisitos RELAX, e posterior criação do modelo conceitual, ambas atividades executadas nas etapas da ER. Analistas normalmente realizam a modelagem conceitual manualmente, o que torna o processo lento e suscetível a erros, além de que seus resultados são dependentes da experiência do modelador. A automação dessa tarefa atua em todos esse pontos: Ela reduz o tempo em que os desenvolvedores gastam para criar o modelo, o que diminui custo; Nossa abordagem automatizada diminui a probabilidade de erro humano, como omissão de informações por exemplo; E completando, nossa abordagem sistemática de criação dos modelos auxilia a diminuir o problema da dependência nos resultados.

Para a comunidade acadêmica, nossa principal contribuição está nas regras de extração que criamos, mas além disso, apresentamos conceitos e ferramentas importantes na área de processamento de linguagens, os quais podem ser utilizadas para outros fins, como a criação de uma nova linguagem de programação/especificação ou então métodos de extração de informações de textos em LN.

7.2 Trabalhos Futuros

Para trabalhos futuros, pretendemos adicionar a possibilidade de processamento em lote dos requisitos, atualmente processamos apenas um, e assim criar modelos mais completos. Pretendemos também adicionar funcionalidades para ilustrar visualmente o modelo extraído utilizando EMF do Eclipse. E ainda, adicionar uma opção para exportar o modelo no formato *XML Metadata Interchange* (XMI), o que possibilita que outras ferramentas de modelagem visualizem nosso modelo, tornando nossas saídas intercambiáveis.

Além disso, pretendemos avaliar nossa ferramenta com um *corpus* maior de requisitos, o que possivelmente pode apontar limitações, as quais devem alimentar nossas regras de extração, tornando-as mais robustas. E por fim, pretendemos verificar nos mo-

delos extraídos, padrões de relacionamentos que podem ser resolvidos com a aplicação de padrões de projeto.

REFERÊNCIAS

AFREEN, H.; BAJWA, I. S.; BORDBAR, B. Sbvr2uml: A challenging transformation. In: IEEE. **Frontiers of Information Technology (FIT), 2011**. [S.l.], 2011. p. 33–38. Citado 8 vezes nas páginas 25, 51, 52, 54, 55, 56, 73 e 82.

AHO, A.; ULLMAN, J.; SETHI, R. **COMPILADORES: PRINCIPIOS, TECNICAS E FERRAMENTAS**. [S.l.]: LTC, 2008. ISBN 9788521610571. Citado 2 vezes nas páginas 31 e 32.

BENDER, E. **Mathematical Methods in Artificial Intelligence**. [S.l.]: Wiley, 1996. (IEEE Computer Society Press). ISBN 9780818672002. Citado na página 30.

BETTINI, L. **Implementing domain-specific languages with Xtext and Xtend**. [S.l.]: Packt Publishing Ltd, 2016. Citado 4 vezes nas páginas 31, 32, 42 e 43.

BIRD, S.; KLEIN, E.; LOPER, E. **Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit**. [S.l.]: O'Reilly Media, 2009. ISBN 9780596555719. Citado 2 vezes nas páginas 35 e 36.

BOLLOJU, N.; SCHNEIDER, C.; SUGUMARAN, V. A knowledge-based system for improving the consistency between object models and use case narratives. **Expert Systems with Applications**, Elsevier, v. 39, n. 10, p. 9398–9410, 2012. Citado 5 vezes nas páginas 51, 52, 54, 55 e 57.

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. Citado 3 vezes nas páginas 23, 24 e 29.

BRUN, Y. et al. Engineering self-adaptive systems through feedback loops. In: **Software engineering for self-adaptive systems**. [S.l.]: Springer, 2009. p. 48–70. Citado na página 27.

COCKBURN, A. **Escrevendo Casos de Usos Eficazes: Um guia prático para desenvolvedores de software**. [S.l.]: Bookman, 2005. ISBN 9788577800193. Citado na página 60.

DAHHANE, W. et al. An automated object-based approach to transforming requirements to class diagrams. In: IEEE. **Complex Systems (WCCS), 2014 Second World Conference on**. [S.l.], 2014. p. 158–163. Citado 4 vezes nas páginas 51, 52, 55 e 57.

DEEPTIMAHANTI, D. K.; SANYAL, R. Semi-automatic generation of uml models from natural language requirements. In: ACM. **Proceedings of the 4th India Software Engineering Conference**. [S.l.], 2011. p. 165–174. Citado 6 vezes nas páginas 24, 51, 52, 54, 55 e 56.

ELBENDAK, M.; VICKERS, P.; ROSSITER, N. Parsed use case descriptions as a basis for object-oriented class model generation. **Journal of Systems and Software**, Elsevier, v. 84, n. 7, p. 1209–1223, 2011. Citado 10 vezes nas páginas 25, 51, 52, 54, 55, 57, 73, 74, 81 e 82.

FATWANTO, A. Software requirements translation from natural language to object-oriented model. In: **2012 IEEE Conference on Control, Systems & Industrial Informatics**. [S.l.: s.n.], 2012. Citado 5 vezes nas páginas 51, 52, 54, 56 e 57.

FERREIRA, D. de A.; SILVA, A. Rodrigues da. Rsl-il: An interlingua for formally documenting requirements. In: IEEE. **Model-Driven Requirements Engineering (MoDRE), 2013 International Workshop on**. [S.l.], 2013. p. 40–49. Citado 5 vezes nas páginas 51, 52, 53, 56 e 57.

FOUNDATION, E. Eclipse modeling framework (emf) - reference documentation. Disponível em: <https://www.eclipse.org/modeling/emf/>. Acesso em: 02 de junho de 2017. 2017. Citado na página 40.

FOUNDATION, E. Xtext - reference documentation. Disponível em: <https://eclipse.org/Xtext/documentation/index.html>. Acesso em: 02 de junho de 2017. 2017. Citado na página 42.

JUNIOR, O. A. de A. Engenharia de requisitos para sistemas auto-adaptativos. 2013. Citado 2 vezes nas páginas 23 e 27.

JURAFSKY, D.; MARTIN, J. **Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition**. [S.l.]: Pearson Prentice Hall, 2009. (Prentice Hall series in artificial intelligence). ISBN 9780131873216. Citado 5 vezes nas páginas 32, 33, 34, 35 e 36.

JYOTHILAKSHMI, M.; SAMUEL, P. Domain ontology based class diagram generation from functional requirements. In: IEEE. **Intelligent Systems Design and Applications (ISDA), 2012 12th International Conference on**. [S.l.], 2012. p. 380–385. Citado 5 vezes nas páginas 51, 52, 53, 55 e 57.

LEMOS, R. D. et al. Software engineering for self-adaptive systems: a second research roadmap. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. **Dagstuhl Seminar proceedings**. [S.l.], 2011. Citado na página 23.

LEVY, R.; ANDREW, G. Tregex and tsurgeon: tools for querying and manipulating tree data structures. In: CITESEER. **Proceedings of the fifth international conference on Language Resources and Evaluation**. [S.l.], 2006. p. 2231–2234. Citado na página 45.

LIU, L.; LI, T.; KOU, X. Eliciting relations from natural language requirements documents based on linguistic and statistical analysis. In: IEEE. **Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual**. [S.l.], 2014. p. 191–200. Citado 5 vezes nas páginas 51, 52, 53, 55 e 57.

MACÍAS-ESCRIVÁ, F. D. et al. Self-adaptive systems: A survey of current approaches, research challenges and applications. **Expert Systems with Applications**, Elsevier, v. 40, n. 18, p. 7267–7279, 2013. Citado 2 vezes nas páginas 23 e 27.

MANNING, C. D. et al. The stanford corenlp natural language processing toolkit. In: **ACL (System Demonstrations)**. [S.l.: s.n.], 2014. p. 55–60. Citado na página 44.

MARNEFFE, M.-C. D. et al. Universal stanford dependencies: A cross-linguistic typology. In: **LREC**. [S.l.: s.n.], 2014. v. 14, p. 4585–92. Citado na página 36.

MORO, G. B. Uma ferramenta de apoio à especificação de requisitos para sistemas autoadaptativos. 2015. Citado 6 vezes nas páginas 25, 68, 69, 71, 72 e 83.

PETERSEN, K. et al. Systematic mapping studies in software engineering. In: SN. **12th international conference on evaluation and assessment in software engineering**. [S.l.], 2008. v. 17, n. 1, p. 1–10. Citado na página 49.

PFLEEGER, S.; ATLEE, J. **Software Engineering: Theory and Practice**. [S.l.]: Prentice Hall, 2004. v. 2. Citado na página 24.

PRESSMAN, R. **Engenharia de Software**. McGraw Hill Brasil, 2011. ISBN 9788580550443. Disponível em: <http://books.google.com.br/books?id=y0rH9wuXe68C>. Citado 3 vezes nas páginas 24, 29 e 30.

RIVIÊRES, J. des; WIEGAND, J. Eclipse: A platform for integrating development tools. **IBM Systems Journal**, International Business Machines Corporation, v. 43, n. 2, p. 371, 2004. Citado na página 39.

RUSSELL, S.; NORVIG, P. **Inteligência artificial**. [S.l.]: CAMPUS - RJ, 2004. ISBN 9788535211771. Citado 2 vezes nas páginas 30 e 35.

SAGAR, V. B. R. V.; ABIRAMI, S. Conceptual modeling of natural language functional requirements. **Journal of Systems and Software**, Elsevier, v. 88, p. 25–41, 2014. Citado 9 vezes nas páginas 25, 51, 52, 53, 55, 56, 73, 74 e 82.

SALEHIE, M.; TAHVILDARI, L. Self-adaptive software: Landscape and research challenges. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, ACM, v. 4, n. 2, p. 14, 2009. Citado 2 vezes nas páginas 27 e 28.

SCHUSTER, S.; MANNING, C. D. Enhanced english universal dependencies: An improved representation for natural language understanding tasks. In: **Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)**. [S.l.: s.n.], 2016. Citado 2 vezes nas páginas 35 e 36.

SHAKYA, B.; NANTAJEEWARAWAT, E. Towards generation of sequence diagrams from operation contracts and design patterns. In: IEEE. **Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2013 10th International Conference on**. [S.l.], 2013. p. 1–6. Citado 5 vezes nas páginas 51, 52, 53, 56 e 58.

SHARMA, R.; SRIVASTAVA, P. K.; BISWAS, K. K. From natural language requirements to uml class diagrams. In: IEEE. **Artificial Intelligence for Requirements Engineering (AIRE), 2015 IEEE Second International Workshop on**. [S.l.], 2015. p. 1–8. Citado 10 vezes nas páginas 24, 25, 51, 52, 55, 56, 73, 74, 81 e 82.

VOGEL, L.; MILINKOVICH, M. **Eclipse 4 RCP: The Complete Guide to Eclipse Application Development;[Based on Eclipse 4.3]; Version 8.1**. [S.l.]: Vogella, 2013. Citado 3 vezes nas páginas 39, 40 e 41.

WAZLAWICK, R. S. **Análise e Projeto de Sistemas da Informação**. [S.l.]: Elsevier Brasil, 2004. v. 2. Citado 2 vezes nas páginas 23 e 29.

WHITTLE, J. et al. Relax: a language to address uncertainty in self-adaptive systems requirement. **Requirements Engineering**, Springer, v. 15, n. 2, p. 177–196, 2010. Citado 7 vezes nas páginas 23, 27, 45, 46, 47, 48 e 69.

YUE, T.; BRIAND, L. C.; LABICHE, Y. atoucan: An automated framework to derive uml analysis models from use case models. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM, v. 24, n. 3, p. 13, 2015. Citado 4 vezes nas páginas 51, 52, 55 e 56.

Apêndices

APÊNDICE A – GRAMÁTICA DA RELAX EM *GRAMMAR-LANGUAGE*

```
/**
* @author Luiz Paulo Franz <luizpaulofranz at gmail.com>
* @version 1.0
* @since version 1.0 2017-01
*/

grammar br.edu.unipampa.lesa.relaxEditor.Relax with org.eclipse.xtext.common.Terminals
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate relax "http://www.edu.br/unipampa/lesa/relaxEditor/Relax"

//contem todos os requisitos, no raiz
Requirements:
requirements+=Requirement+;

//declaramos tipos ainda nao existentes
terminal BOOLEAN returns ecore::EBoolean:"true"|"false";

//sobrescrevemos a regra nativa ID, para acrescentar um caractere de escape "\"
terminal ID: "\\"?("a".."z"|"A".."Z"|"_") ("a".."z"|"A".."Z"|"_"|"0".."9")*;


terminal STRING : '"' ( "\\" . /* "b"|"t"|"n"|"f"|"r"|"u"|'"'|"'"|"\\" */ | !("\\"|'"') )* '"';

//Natural Language, trechos que serao em LN
NL hidden(WS):(ID|INT|","|"'"|"("|")")+;

//cada requisito RELAX inicia com "REQ ou req" e finaliza com o ";"
Requirement:
("req"|"REQ") name=ID ":" (root+=Sentence)+ ";" (properties+=Properties)*;

//cada sentenca eh finalizado com o ".", requisitos podem conter varias sentencas
```

```
32 Sentence:
33 (root+=General)+ & '.'?;
34
35 //propriedades do ambiente
36 Properties: Envs|Mons|Rels|Dep;
37
38 Envs: "ENV" "{" (envs+=Env)+ "}";
39 Mons: "MON" "{" (mons+=Mon)+"}";
40 Rels: "REL" "{" (rels+=Rel)+ "}";
41 Dep: "DEP:" dep+=[Requirement] ("," dep+=[Requirement])* ";";
42
43 Env: name=ID ":" value=NL";";
44 Mon: name=ID ":" value=NL";";
45 Rel: mon=[Mon]"-"env=[Env]";";
46
47 //tratando a recursividade a esquerda causada pelo UNTIL
48 General:
49 rest = Rest =>({UntilOperator.left=current}('UNTIL'|'until')
      right=General)*;
50
51 //aqui se encontram as demais regras da gramatica
52 Rest: rule = Primitive|ShallOperator|MayOperator|
      EventuallyOperator|BeforeOperator|AfterOperator|InOperator|
      AsCloseOperator|AsOperator;
53
54 // tipos primitivos, como boleanos (previstos na gramatica
      original) e os trechos em LN
55 Primitive:
56 BooleanValue | StringValue;
57
58 //novamente os operadores primitivos
59 BooleanValue:value=BOOLEAN;
60 StringValue:value=NL;
61
62 //daqui para baixo sao tratados todos os operandos (exceto o
      UNTIL)
63 ShallOperator:
64 ("SHALL"|"shall") elements=General;
65
66 MayOperator:
67 ("MAY"|"may") action1=General ("OR" "MAY"|"or" "may") action2=
      General;
```

```
68
69 EventuallyOperator:
70 ("EVENTUALLY"|"eventually") element=General;
71
72 BeforeOperator:
73 ('BEFORE'|'before') event=NL;
74
75 AfterOperator:
76 ("AFTER"|"after") event=NL;
77
78 InOperator:
79 ("IN"|"in") t=NL;
80
81 //Frquencia ou Quantidade f or q
82 AsCloseOperator:
83 ("AS" "CLOSE" "AS" "POSSIBLE" "TO"|"as" "close" "as" "possible" "
      to") ForQ=NL;
84
85 //quebramos todos os operadores AS
86 AsOperator:
87 AsEarlyOperator|AsLateOperator|AsManyOperator|AsFewOperator;
88
89 //usamos a variavel "op" para identificar qual e o operador
90 AsEarlyOperator:
91 ("AS" op="EARLY" "AS" "POSSIBLE"|"as" op="early" "as" "possible")
      ;
92
93 AsLateOperator:
94 ("AS" op="LATE" "AS" "POSSIBLE"|"as" op="late" "as" "possible");
95
96 //perceba que a variavel "element" eh opctional
97 AsManyOperator:
98 ("AS" op="MANY"|"as" op="many") (element=NL)? ("AS" "POSSIBLE"|"
      as" "possible");
99 AsFewOperator:
100 ("AS" op="FEW"|"as" op="few") (element=NL)? ("AS" "POSSIBLE"|"as"
       "possible");
```

B REQUISITOS DO CENÁRIO *SMARTCAR* DESCRITOS COM A RELAX EDITOR

```
REQ car01: The smartcar SHALL detect and notify mechanical
   problems;

ENV{engine: Smartcar's engine;}
MON{engineSensor: Sensor \in car's engine;}
REL{engineSensor-engine;}
```

```
REQ car02: The Smartcar MAY turn on air conditioner OR MAY open
   the windows;

ENV{temperature: Smartcar's internal temperature;}
MON{temperatureSensor: Thermometer \to monitoring the Smartcar's
   temperature;}
REL{temperatureSensor-temperature;}
```

```
REQ car03: The Smartcar EVENTUALLY recharge your batteries;

ENV{battery: The Smartcar's battery;}
MON{batterySensor: A sensor \to identify the battery level;}
REL{batterySensor - battery;}
```

```
REQ car04: The Smartcar SHALL monitor location UNTIL the car stop
   ;

ENV{currentLocation: The location of Smartcar;}
MON{gps: GPS on the Smartcar;}
REL{gps - currentLocation;}
```

```
REQ car05: The Smartcar SHALL check the road condiction BEFORE
   starting the trip;

ENV{road: The road where the Smartcar will pass;}
MON{camera: Cameras \to monitoring the road;}
REL{camera - road;}
```

```
REQ car06: The Smartcar SHALL check the car condiction AFTER
   finishing the trip;

ENV{engine: Smartcar's engine;}
MON{engineSensor: Sensor \in car's engine;}
REL{engineSensor-engine;}
```

```
REQ car07: The Smartcar SHALL seek and record all service points
   IN 1 hour period AFTER arriving \in a new city;

ENV{servicePoints: City service points;}
MON{serviceSensor: Diferent services sensors;}
REL{serviceSensor - servicePoints;}
```

```
REQ car08: The Smartcar SHALL inform the existence of new emails
   AS EARLY AS POSSIBLE AFTER connecting \to the wireless network;

ENV{mail: The configured mail box;}
MON{network: The Smartcar wifi network sensor;}
REL{network - mail;}
```

```
REQ car09: The Smartcar SHALL inform the need for oil change AS
   LATE AS POSSIBLE AFTER a previous oil change;

ENV{oil: Engine's oil;}
MON{oilSensor: A sensor \to monitoring oil;}
REL{oilSensor - oil;}
```

```
REQ car10: The Smartcar SHALL notify need \to make a car checkup
   AS CLOSE AS POSSIBLE TO 12 months \or AS CLOSE AS POSSIBLE TO
   1500 Km AFTER last car checkup;

ENV{car: The Smartcar itself;}
MON{carSensors: Sensors instaled \in Smartcar;}
REL{carSensors - car;}
```

```
REQ car11: The Smartcar SHALL calculate AS MANY alternative
   routes AS POSSIBLE AFTER starting the trip;

ENV{map: The trip's map;}
MON{gps: GPS on Smartcar;}
REL{gps - map;}
```

```
REQ car12: The Smartcar SHALL calculate AS FEW refueling point AS
    POSSIBLE AFTER starting the trip;

ENV{map: The trip map;}
MON{serviceSensor: Diferent services sensors;}
REL{serviceSensor - map;}
```

C MODELOS CONCEITUAIS DO CENÁRIO *SMARTCAR*

Figura 20 – Modelo conceitual RelaxML do requisito car01.

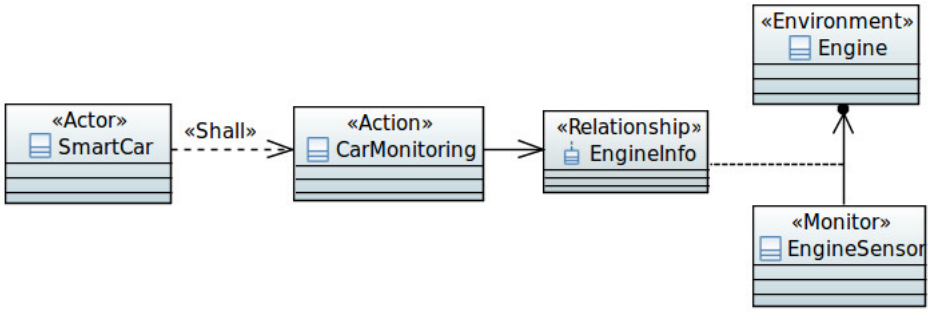


Figura 21 – Modelo conceitual RelaxML do requisito car02.

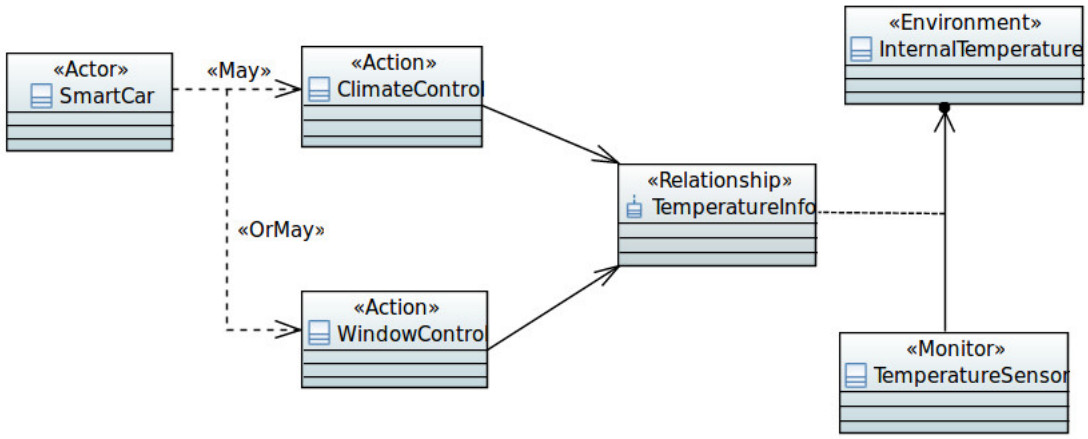


Figura 22 – Modelo conceitual RelaxML do requisito car03.

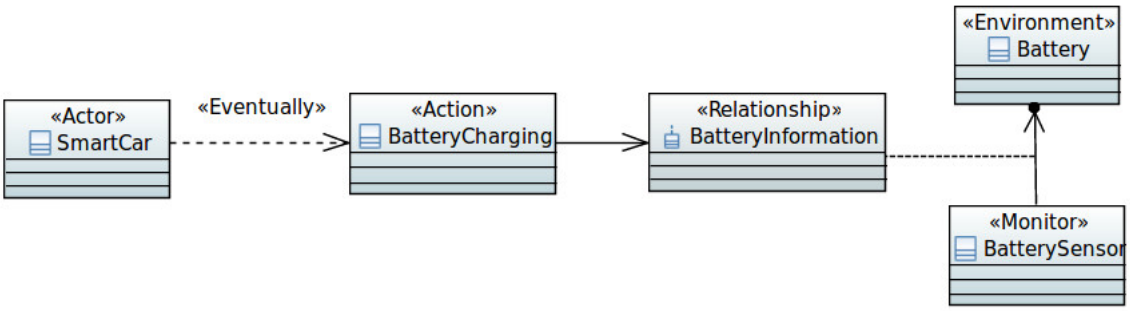


Figura 23 – Modelo conceitual RelaxML do requisito car04.

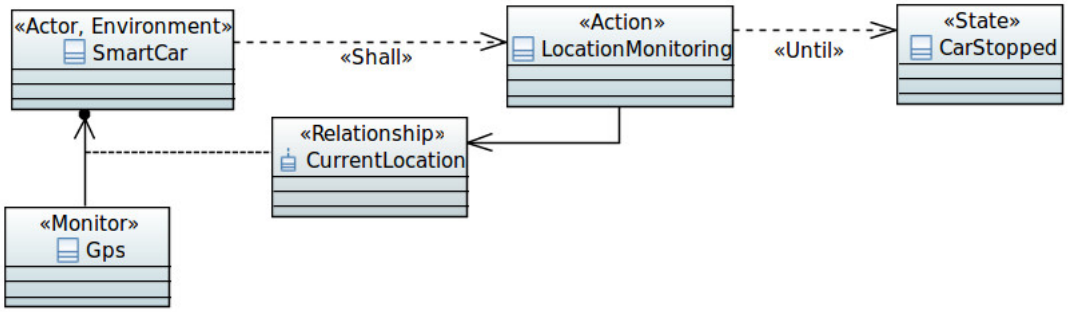


Figura 24 – Modelo conceitual RelaxML do requisito car05.

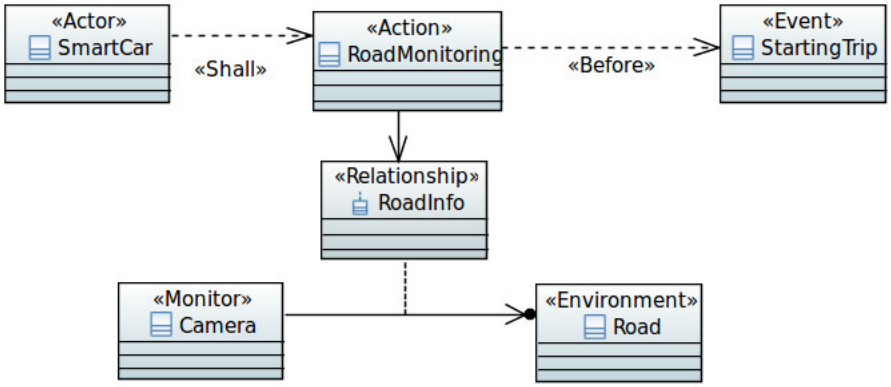


Figura 25 – Modelo conceitual RelaxML do requisito car06.

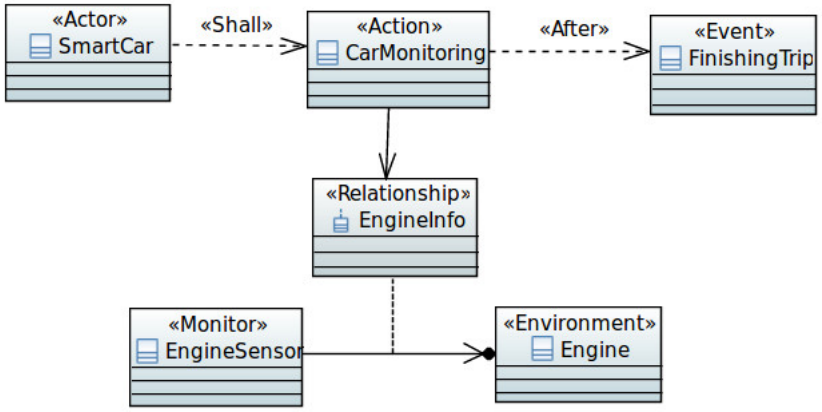


Figura 26 – Modelo conceitual RelaxML do requisito car07.

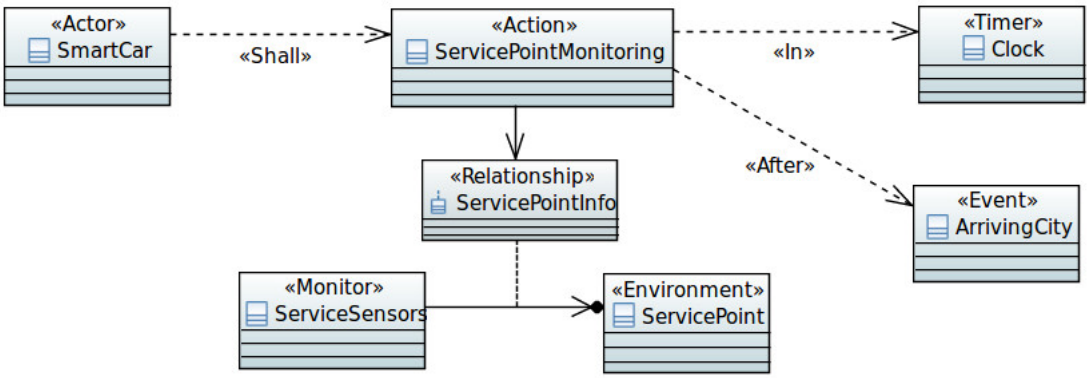


Figura 27 – Modelo conceitual RelaxML do requisito car08.

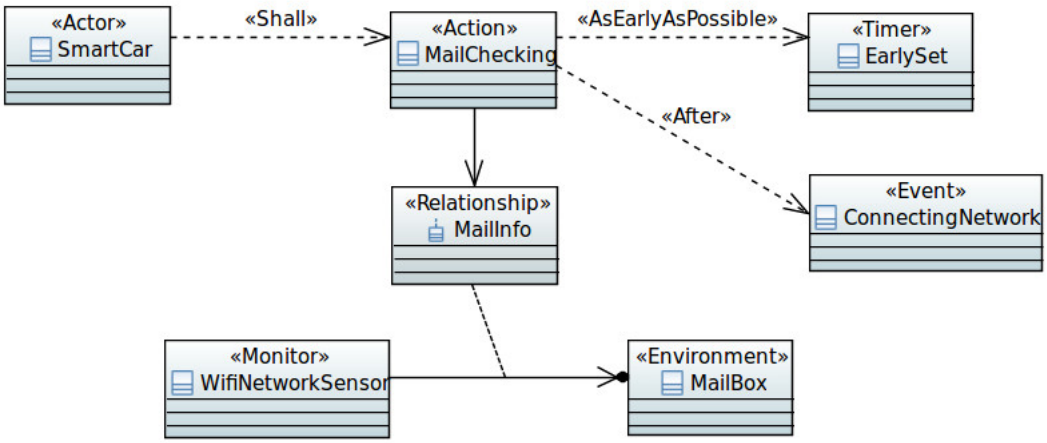


Figura 28 – Modelo conceitual RelaxML do requisito car09.

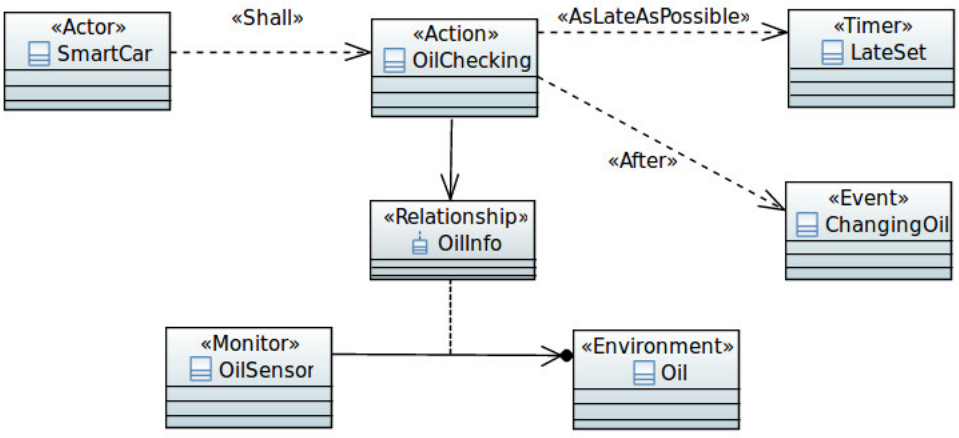


Figura 29 – Modelo conceitual RelaxML do requisito car10.

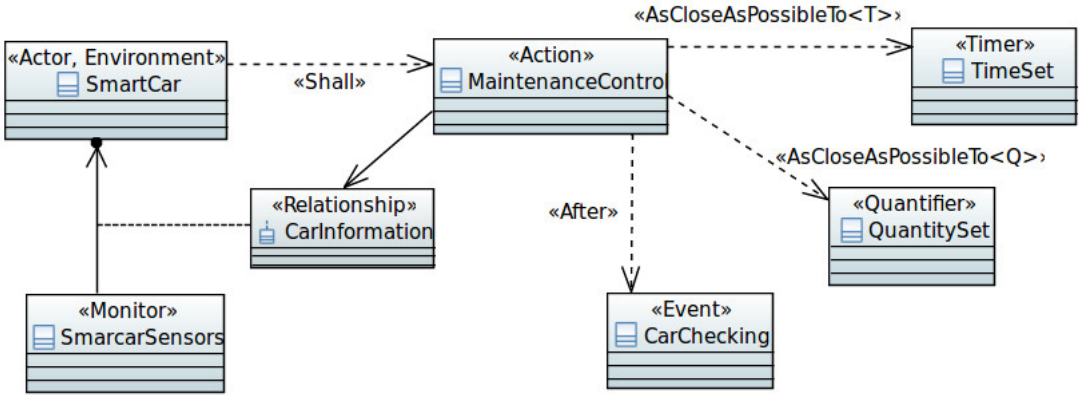


Figura 30 – Modelo conceitual RelaxML do requisito car11.

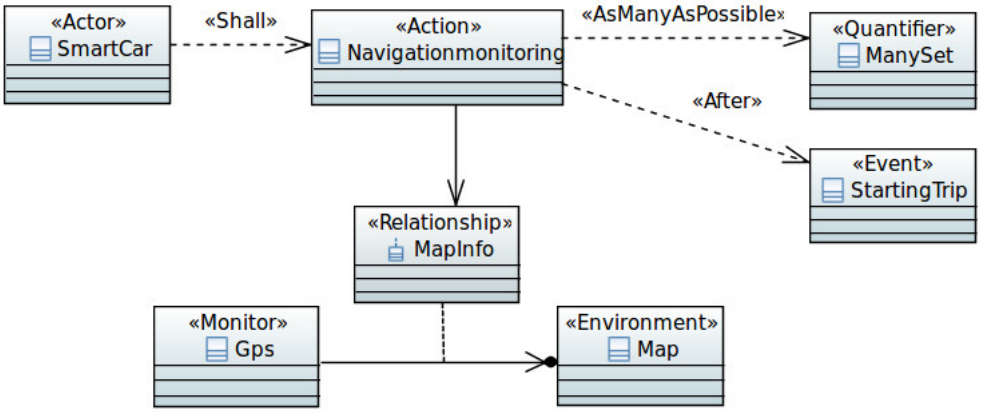


Figura 31 – Modelo conceitual RelaxML do requisito car12.