UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCOS ANTONIO DE OLIVEIRA JUNIOR

Especificação e Análise de Sistemas através de Gramática de Grafos

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Profª. Drª. Leila Ribeira
Coorientadora: Profª. Drª. Érika Fernandes Cota

Porto Alegre
2016

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Oliveira Jr., Marcos Antonio de
Especificação e Análise de Sistemas através de Gramática de Grafos / Marcos Antonio de Oliveira Junior. – 2016.
116 f.:il.
Orientadora: Leila Ribeiro; Coorientadora: Érika Fernandes Cota.
Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2016.
1.Gramática de Grafos. 2.Extração de Modelos a partir de código. 3.Caso de Uso. I. Ribeiro, Leila. II. Cota, Érika Fernandes. III. Especificação e Análise de Sistemas através de Gramática de Grafos.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“The mind that opens to a new idea
never returns to its original size.”
Albert Einstein

AGRADECIMENTOS

Primeiramente, agradeço a Deus pela minha vida, pela família com a qual fui agraciado e pelas oportunidades que tive em minha vida. Agradeço muito aos meus pais, Marcos e Denise, os quais me deram total apoio, não somente no mestrado, mas em todos os momentos da minha vida, em todas dificuldades, sempre muito presentes e compreensivos, me instruindo e fazendo com que eu conseguisse suportar e superar a tensão, os medos e os problemas. Agradeço também aos demais familiares e amigos, por saber compreender os momentos de ausência e pelo carinho, por todas as vezes que fizeram com que meus dias ficassem melhores, mais animados, e sempre deram força pra seguir a caminhada. Em especial, agradeço às professoras Leila e Érika, pela orientação nesse trabalho, dedicação e paciência que tiveram comigo, e também ao professor Lúcio, o qual teve grande participação nesse trabalho. Agradeço também os demais professores do Instituto de Informática da UFRGS, pelo conhecimento transmitido durante esse período, não somente dentro da sala de aula, como também como amigos e profissionais da área. Por fim, deixo meu agradecimento aos colegas do PPG Computação, das disciplinas e do laboratório, pela acolhida e pelo ambiente de trabalho criado, proporcionando sempre uma grande troca de conhecimentos. Muito obrigado a todos! Um grande abraço!

RESUMO

O crescimento da complexidade e do tamanho dos sistemas computacionais atuais suscitou um aumento na dificuldade de extração e especificação de modelos formais desses sistemas, tornando essa atividade cada vez mais dispendiosa, tanto em tempo quanto em custo. Modelos são utilizados em diversas técnicas da Engenharia de Software, com o intuito de auxiliar em processos que compreendem desde o desenvolvimento de novos softwares, até reconstrução de um sistema a partir de software legado, passando pela realização de manutenção de um software em operação. Portanto, é necessário que essas abstrações sejam confiáveis e representem fielmente o software real. Nesse sentido, a adoção de métodos formais para a construção e análise de modelos computacionais é crescente e motivada, principalmente, pela confiabilidade que os formalismos matemáticos agregam aos modelos. No entanto, a utilização de métodos formais geralmente demanda um alto investimento de recursos humanos e, consequentemente, financeiros, uma vez que a utilização de tais formalismos é condicionada ao estudo profundo de sua fundamentação matemática. Considerando-se a extensa aplicabilidade de modelos em diversas subáreas da Ciência da Computação e as vantagens advindas da utilização de métodos formais para especificar sistemas, é interessante identificar métodos e ferramentas existentes para automatizar os processos de extração e análises de modelos, em conjunto com a adoção de formalismos que possam ser utilizados por profissionais da computação que atuam na indústria de software. Dessa forma, é estimulada nesse trabalho a utilização do formalismo de Gramática de Grafos, um método formal que diferencia-se dos demais por ser intuitivo e possuir uma representação visual gráfica, o que facilita a sua compreensão e não exige um conhecimento avançado sobre o formalismo. Primeiramente, é proposta uma abordagem para a extração de modelos em Gramática de Grafos a partir de código-fonte, extraindo informações de execuções de código Java anotado. Em seguida, é apresentada uma metodologia existente para extração e análise de Gramática de Grafos a partir de Casos de Uso, juntamente com um estudo empírico realizado a fim de validar a metodologia. Por fim, são propostas possíveis verificações adicionais, a fim de extender as análises dessa metodologia. Com isso, busca-se a obtenção de modelos, descritos através do formalismo de grafos, a partir de artefatos criados nos dois pólos do processo de desenvolvimento de software, antes e depois da implementação, no sentido de viabilizar futuras comparações, no contexto de verificação de software.

Palavras-chave: Gramática de Grafos. Extração de modelos a partir de código. Caso de Uso.

Specification and Analysis Systems through Graph Grammars

ABSTRACT

The growing size and complexity of current computer systems leading to an increase in the difficulty of extraction and specification of formal models of such systems, making it increasingly expensive activity, both in time and in cost. Models are used in various techniques of software engineering in order to assist in processes that range from the development of new software, to rebuild a system from legacy software, passing for performing maintenance of software in operation. Therefore, it is necessary that these abstractions are reliable and faithfully represent the actual software. In this sense, the adoption of formal methods for the construction and analysis of models is growing and motivated mainly by the reliability that the mathematical formalism add to models. However, the use of formal methods generally demands a high investment in human resources and hence financial, since the use of such formalism is conditioned to the deep study of its mathematical foundation. Considering the extensive applicability of models in various subfields of computer science and the benefits arising from the use of formal methods for specifying systems, it is interesting to identify existing methods and tools to automate the process of extracting models, in addition to the adoption of formalism that can be used by computer professionals working in the software industry. Thus, we encourage the use of the Graph Grammar formalism, a formal method that differs from others because it is intuitive and has a graphical visual representation, making it easy to understand and does not require an advanced knowledge of the formalism. First, we propose an approach for extracting models from source code in Graph Grammar, getting information of executions of annotated Java code. Then an existing methodology for extraction and analysis of Graph Grammar from Use Cases is presented, along with an empirical study to validate the methodology. Finally, we propose possible additional checks in order to extend the analysis of this methodology. Thus, this work aims to extract models, described by the formalism of graphs, from artifacts created in the two poles of the software development process, before and after implementation, in order to allow future comparisons, in the context of software verification.

Keywords: Graph Grammar. Models extraction from source code. Use case.

LISTA DE FIGURAS

Figura 2.1 – Ciclo de desenvolvimento de software. 21
Figura 2.2 – Visão geral da técnica de Verificação de Modelos (Model Checking). 25
Figura 3.1 – Exemplos de Grafos: (a) grafo simples não direcionado; (b) dígrafo ou grafo direcionado; (c) grafo com pesos associados às arestas. 27
Figura 3.2 – Exemplos da aplicação de grafos na especificação de relacionamentos: (a) análise comportamental de usuários de aplicativos para dispositivos móveis; (b) representação de adjacências e distâncias entre territórios. 28
Figura 3.3 – Exemplo de elementos da sintaxe de uma Gramática de Grafos: (a) um grafo *G* tipado, representando um estado sistema, que pode ser inicial ou não, construído de acordo com um grafo-tipo; (b) um grafo-tipo do sistema que representa todos os tipos de arestas e vértices existentes no sistema. 30
Figura 3.4 – (a) morfismo de grafos; (b) não é morfismo de grafos. 31
Figura 3.5 – Exemplo de uma gramática de grafos para um sistema cliente/servidor: grafo inicial (*G0*), grafo-tipo (*T*) e o conjunto de regras (*sendMSG*, *getDATA*, *receiveMSG* e *deleteMSG)*. 32
Figura 3.6 – Exemplo de *Passo de Derivação*: Aplicação de uma regra $r : L \rightarrow R$ a um grafo G. 33
Figura 3.7 – Exemplo de uma regra concorrente gerada a partir da Sequência de Regras *SR1*. 35
Figura 3.8 – Exemplo de conflito no sistema cliente/servidor. 36
Figura 3.9 – Exemplo de matriz de conflitos entre as regras do sistema cliente/servidor gerada através da ferramenta AGG. 37
Figura 3.10 – Exemplo de dependência no sistema cliente/servidor. 38
Figura 3.11 – Exemplo de matriz de dependências entres as regras do sistema cliente/servidor gerada através da ferramenta AGG. 39
Figura 4.1 – Modelo de regra da gramática de grafos utilizada. 40
Figura 4.2 – Fluxograma do processo realizado. 41
Figura 4.3 – Exemplo de uma tabela de Contextos. 43
Figura 4.4 – Exemplo de um arquivo de registro de execução. 44
Figura 4.5 – Fragmento de um Rastro de Contexto. 45
Figura 4.6 – Algoritmo para construir uma gramática de grafos a partir de contextos. 46
Figura 4.7 – Exemplo do funcionamento do algoritmo para mapeamento dos contextos para regras da gramática de grafos, nos dois primeiros casos do algoritmo. 47
Figura 4.8 – Exemplo do funcionamento do algoritmo para mapeamento dos contextos para regras da gramática de grafos quando existem múltiplas ações entre um par de contextos. 48
Figura 4.9 – Código-fonte Java da classe *TrafficLights*. 49
Figura 4.10 – Código-fonte Java da classe *AirConditioner*, uma das quatro classes da aplicação. 50
Figura 5.1 – Visão geral da estratégia para formalização e verificação de casos de uso através do formalismo de transformação de grafos. 59
Figura 5.2 – Caso de uso *Login*, utilizado como exemplo. 60
Figura 5.3 – Resultado da aplicação do Passo 6.2 ao caso de uso *Login*. 63
Figura 5.4 – Resultado da aplicação do Passo 6.3 ao caso de uso *Login*. 63
Figura 5.5 – Descrições do caso de uso antes (esquerda) e depois (direita) da aplicação da metodologia para formalização e verificação de casos de uso. 64
Figura 5.6 – Definição de um *Object Flow* para a Sequência de Regras que representa execução básica do Caso de Uso *Login*. 76
Figura 5.7 – Regra Concorrente Máxima extraída da Sequência de Regras que representa execução básica do Caso de Uso *Login*. 78
Figura 5.8 – Regra *validate&display*, referente ao passo 5 do Caso de Uso *Login*. 79
Figura 5.9 – Regra Concorrente gerada através de combinações entre as regras da Sequência de Regras que representa execução básica do Caso de Uso *Login*. 80

LISTA DE TABELAS

Tabela 4.1 - Resultados da extração de Gramática de Grafos para os casos de estudo. 51
Tabela 4.2 - Resultado da extração de Gramática de Grafos após a aplicação das estratégias de otimização do algoritmo. 53
Tabela 5.1 - Definição de objetivos do estudo, segundo o modelo GQM. 65
Tabela 5.2 - Resultados do estudo empírico. 70
Tabela 5.3 - Resultados preliminares da análise de concorrência em Casos de Uso. 82

LISTA DE ABREVIATURAS E SIGLAS

AGG	The Attributed Graph Grammar System
CFG	Control FLow Graph
DSL	Domain-Specific Language
GG	Gramática de Grafos
LHS	Left-Hand Side
MBT	Model-Based Testing
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
RC	Regra Concorrente
RHS	Right-Hand Side
SR	Sequência de Regras

SUMÁRIO

1 INTRODUÇÃO 11
1.1 Motivação 15
1.2 Objetivos 17
2 MODELOS 19
2.1 Definição 19
2.2 Modelos na Engenharia de Software 19
2.2.1 Engenharia de Software Progressiva 20
2.2.2 Engenharia de Software Reversa 21
2.3 Extração de Modelos 22
2.3.1 Modelos baseados em Informações Estáticas 22
2.3.2 Modelos baseados em Informações Dinâmicas 23
2.3.3 Modelos baseados em Informações Híbridas 24
2.4 Engenharia Dirigida por Modelos 24
2.5 Verificação de Modelos 25
3 GRAMÁTICA DE GRAFOS 27
3.1 Grafo 27
3.2 Definição de Gramática de Grafos 28
3.2.1 Sintaxe de Gramática de Grafos 29
3.2.1.1 Grafo Inicial **29**
3.2.1.2 Grafo-tipo **30**
3.2.1.3 Regras **30**
3.2.1.4 Exemplo de Gramática de Grafo **31**
3.2.2 Semântica de Gramática de Grafos 33
3.2.2.1 Passo de Derivação **33**
3.3 Regras Concorrentes 34
3.4 Pares Críticos 35
3.4.1 Análise de Conflitos 35
3.4.2 Análise de Dependências 38
4 EXTRAÇÃO DE GRAMÁTICA DE GRAFOS A PARTIR DE CÓDIGO 40
4.1 Metodologia 41
4.1.1 Extração do Modelo 42
4.1.1.1 Tabela de Contextos **42**
4.1.1.2 Arquivos de Registros de Execução **43**
4.1.1.3 Rastos de Contextos **44**
4.1.1.4 Geração da Gramática de Grafos **45**
4.1.2 Casos de Estudo 48
4.1.2.1 Traffic Lights **48**
4.1.2.2 Air Conditioner **49**
4.1.3 Experimentos Preliminares 51
4.1.4 Otimizações na Geração da Gramática de Grafos 51
4.1.4.1 Program Counter **52**

4.1.4.2 Análise de Mensagens **52**
4.1.4.3 Chamadas de Métodos **53**
4.2 Resultados **53**
4.3 Trabalhos Relacionados **54**
5 EXTRAÇÃO DE GRAMÁTICA DE GRAFOS A PARTIR DE CASO DE USO 57
5.1 Metodologia para Formalização de Casos de Uso **58**
5.2 Estudo Empírico **65**
5.2.1 Procedimentos **66**
5.2.1.1 Revisão dos Casos de uso pelo Analista do Sistema **66**
5.2.1.2 Formalização dos Casos de Uso **66**
5.2.1.3 Avaliação dos Possíveis Problemas Encontrados **66**
5.2.1.4 Análise dos Dados **67**
5.2.2 Caso de Estudo **67**
5.2.3 Ameaças à Validade do Estudo Empírico **68**
5.2.4 Resultados **70**
5.3 Análise de Concorrência **72**
5.3.1 Extensão da Metodologia **74**
5.3.1.1 Dependências Parciais – Object Flow **74**
5.3.1.2 Combinação de Sobreposições **77**
5.3.2 Resultados Preliminares **81**
5.4 Trabalhos Relacionados **84**
6 CONCLUSÃO **87**
REFERÊNCIAS **91**
ANEXO A – METODOLOGIA PARA FORMALIZAÇÃO DE CASOS DE USO **96**

1 INTRODUÇÃO

Sistemas computacionais são aplicados em diversas áreas do conhecimento, tanto na indústria quanto no comércio, e cada vez mais indispensáveis para um bom funcionamento das atividades cotidianas de vários segmentos do mercado. Benefícios como velocidade nas transações, automatização de operações e transposição de barreiras geográficas são algumas das razões pelas quais softwares se tornaram elementos fundamentais para o desenvolvimento econômico da sociedade. Devido a essas circunstâncias e às características dos sistemas computacionais, gerou-se uma ampla demanda de software, principalmente na última década, período em que a popularização dos computadores atingiu níveis mundiais e o desenvolvimento de sistemas se fortaleceu, impulsionando o número de softwares no mercado.

De acordo com os fundamentos da Engenharia de Software, o desenvolvimento de um software é dividido em diversas etapas e engloba tarefas que vão desde a documentação de um sistema até a implementação de um produto final. A utilização de estratégias de desenvolvimento é fortemente indicada, principalmente no que se refere a equipes de desenvolvimento, onde princípios básicos como uma boa comunicação e boas práticas de desenvolvimento são essenciais para a eficiência do trabalho realizado. Além disso, também é comum o uso de ferramentas pelos desenvolvedores, que auxiliam na compreensão dos sistemas, na manipulação dos artefatos do projeto e, consequentemente, na implementação das aplicações, atividades primordiais quando se almeja um software de qualidade.

Além do desenvolvimento, caracterizado como etapa inicial da implantação de sistemas, frequentemente, operações sobre sistemas em funcionamento se fazem necessárias. São exemplos de processos realizados após a implementação de um sistema a verificação de propriedades, de acordo com alguma especificação, e atividades de manutenção, que podem ser necessárias durante toda a vida útil de um software. Alterações em um software podem ser feitas com diferentes objetivos como correção de falhas, adição de novas funcionalidades, acompanhamento da evolução tecnológica, entre outros. A manipulação de sistemas, seja ela de caráter corretivo ou evolutivo, não é tarefa trivial da Engenharia de Software, pois realizar operações sobre um software já existente, geralmente, envolvem atividades custosas, que requerem um alto investimento de recursos humanos e, consequentemente, financeiros.

A partir dessas circunstâncias, é comum que em projetos de software sejam utilizadas abstrações para representar um sistema, tanto nas fases iniciais quanto após a produção de versões funcionais do sistema. No início de um projeto de software, em fases anteriores à

implementação, é fortemente recomendado que sejam utilizados mecanismos para documentar o software que será criado a fim de guiar os desenvolvedores e também facilitar a compreensão dos mesmos sobre a lógica da aplicação. Além disso, pode ser desejável a obtenção de uma representação do sistema, após a implementação, para fins de análises sobre o comportamento do software implementado. Dessa forma, é possível observar a importância e a necessidade de se obter representações abstratas de sistemas, uma vez que operações e análises sobre um sistema computacional podem ser realizadas com base nesses modelos abstratos, sem a necessidade de manusear diretamente o software em questão.

Na Engenharia de Software, para se criar abstrações de um sistema computacional são utilizados modelos computacionais. Modelos podem ser manipulados e utilizados como artefatos poderosos em diversas etapas do desenvolvimento de um software. É importante observar que podem existir modelos distintos de um mesmo software, contendo informações diferentes, uma vez que o nível de informação que um modelo possui é variável e depende exclusivamente da sua finalidade. A utilização de modelos facilita tanto o desenvolvimento de novos software quanto a realização de alterações e análises sobre programas existentes, tornando-se assim uma alternativa interessante nos processos da Engenharia de Software.

Modelos podem ser construídos paralelamente ao desenvolvimento de um software ou podem ser extraídos a partir de uma implementação. No decorrer das fases iniciais de desenvolvimento, é possível utilizar modelos para auxiliar na especificação do sistema, enquanto que, após a implementação, um modelo extraído de um software representa o comportamento implementado. Portanto, a extração de modelos é bastante aplicada quando existe um software legado sobre o qual não se tem uma documentação ou conhecimento sobre o seu comportamento.

Em diversas subáreas da Engenharia de Software modelos desempenham papel fundamental nas atividades de um projeto de software. No contexto da *Engenharia Dirigida por Modelos (Model Driven Engineering - MDE)*, modelos descrevem os componentes e comportamentos do sistema (DEURSEN, KLINT, VISSER, 2007), em uma abordagem que reúne linguagens de modelagem e mecanismos de transformações, onde os modelos são representados através de notações visuais ou textuais, de acordo com uma sintaxe e uma semântica previamente definidas. No âmbito de manutenção de software, várias metodologias foram desenvolvidas para facilitar as alterações em um sistema, utilizando técnicas e ferramentas para, a partir dos elementos de um sistema, como código fonte, documentação ou interface, extrair modelos com informações relacionadas aos componentes e suas funcionalidades. No domínio de Teste de Software, existem técnicas e ferramentas que

utilizam modelos no design de testes, como o *Teste Baseado em Modelos (Model-Based Testing - MBT)* (UTTING, LEGEARD, 2007), onde os testes são gerados automaticamente por ferramentas que utilizam algoritmos e estratégias de acordo com as diferentes abordagens utilizadas no MBT, explorando os comportamentos do software de acordo com o modelo existente.

Ainda, modelos podem ser utilizados no campo de Validação e Verificação de Software, através de técnicas como a *Verificação de Modelos (Model Checking)* (CLARKE JR, GRUMBERG, PELED, 1999), que verifica de forma automática através de testes exaustivos se, dado um modelo de estados finitos e uma propriedade desejada do sistema, tal propriedade é verdadeira para o modelo fornecido, explorando sistematicamente todas as possibilidades de comportamento do modelo (HOLZMANN, SMITH,1999). Também é possível comparar modelos dois ou mais modelos, com o intuito de observar se tais artefatos representam um mesmo sistema, por exemplo, verificando se um modelo extraído a partir de código possui os mesmos comportamentos de um modelo criado previamente à codificação, ou seja, um antes e um depois da implementação.

Modelos também são muito utilizados em métodos formais, para o desenvolvimento e análise de softwares. Métodos formais são técnicas baseadas em formalismos matemáticos, que podem ser utilizados em diversos estágios de desenvolvimento de um software, desde a fase de especificação, passando pelo desenvolvimento em si, até a realização de verificações sobre os comportamentos de sistemas, após sua implementação (WING, 1990). A utilização desses métodos em projetos de software é crescente e motivada, principalmente, pela necessidade de se construir sistemas cada vez mais confiáveis e robustos, uma vez que as atividades realizadas nessas técnicas são fundamentadas em análises matemáticas apropriadas. Um exemplo de formalismo que pode ser utilizado tanto para especificação de sistemas quanto para a realização de análises sobre um software é o formalismo de transformação de grafos ou Gramática de Grafos.

O formalismo de Gramática de Grafos (EHRIG, 1997) é composto basicamente por grafos, que representam os estados do sistema, e regras, ou produções (transformações de grafos), que representam as transições entre possíveis estados do sistema. Grafos, por sua vez, são abstrações que permitem a representação de objetos e seus relacionamentos, frequentemente utilizados para a representação de sistemas computacionais (BONDY, MURTY, 2007). Entre os métodos formais, esse formalismo destaca-se por ser um método intuitivo, uma vez que a representação gráfica dos elementos facilita a compreensão dos componentes e comportamentos do sistema. Além disso, esse método não exige um

conhecimento profundo sobre o formalismo, tornando-o acessível a profissionais da computação que não trabalham diretamente com formalismos e métodos matemáticos.

Uma representação de um sistema através de transformações de grafos pode ser criada tanto nas fases iniciais de um projeto, para documentar um comportamento esperado do software, quando após a codificação de um sistema, sendo um modelo formal que reproduz o comportamento do software implementado. Além disso, o formalismo de Gramática de Grafos possibilita a realização de análises sobre um sistema, a fim de se obter informações sobre propriedades, componentes e comportamentos reais do software (EHRIG ET AL., 1999).

Formalismos, como Gramática de Grafos, podem também ser utilizados para a verificação e comparação de modelos, no entanto, para isso é necessário que tenhamos modelos descritos em um mesmo formalismo, extraídos, por exemplo, um a partir de um artefato criado nas fases iniciais de desenvolvimento e outro extraído após a implementação do sistema. Existem abordagens para a extração e verificação de modelos, tanto a partir de código-fonte, como descrito em (CORRADINI, 2004) e (ALSHANQITI, HECKEL, KHAN, 2013), quanto a partir de Casos de Uso, conforme apresentado em (SINNIG, CHALIN, KHENDEK, 2013), (ZHAO, DUAN, 2009) e (SINNIG, CHALIN, KHENDEK, 2009). No entanto, cada processo de extração possui características específicas e, devido a sua natureza e finalidade, atentam para situações particulares ou utilizam formalismos distintos, não sendo, portanto, complementares e inviabilizando a comparação dos modelos extraídos através desses processos.

Para verificar um modelo extraído de código, quanto a sua correção, é necessário compará-lo com um outro modelo, que descreve as características essenciais do sistema que ambos devem representar. Sendo assim, neste trabalho, inicialmente, é proposta uma abordagem para extração de modelos em gramática de grafos a partir de código-fonte. Baseada em um algoritmo, desenvolvido para analisar informações extraídas de execuções de código-fonte anotado, ocorre a geração dos elementos da gramática, para, então, obtermos um modelo extraído de código-fonte, descrito em Gramática de Grafos, do comportamento real do software, após a sua implementação.

Uma vez que existe tal modelo, para verificá-lo, é necessário que exista um outro modelo, descrito também pelo formalismo de Gramática de Grafos, capaz de representar o comportamento esperado desse software, ou seja, previamente à implementação. Para a descrição de sistemas, nas fases iniciais de um projeto de software, geralmente, se utilizam modelagens em Casos de Uso, a fim de retratar os comportamentos possíveis do software, por

isso, neste trabalho é utilizada a metodologia apresentada em Ribeiro et al. (2014), para a extração de modelos descritos em Gramática de Grafos a partir de Casos de Uso. Sendo assim, como sequência deste trabalho, é feita uma validação da metodologia apresentada em Ribeiro et al. (2014), a fim de utilizá-la para a extração de Gramática de Grafo a partir de Casos de Uso, bem como é proposta uma extensão às verificações realizadas pela metodologia, utilizando alguns conceitos de regras concorrentes em Gramática de Grafos. A partir da exploração de modelos descritos através do formalismo de grafos, extraídos de artefatos pré e pós-implementação, tem-se como objetivo possibilitar a comparação desses modelos, um dos trabalhos futuros da pesquisa aqui apresentada.

Esta dissertação é estruturada em capítulos, de forma que: o Capítulo 2 contém uma fundamentação teórica acerca de Modelos e processos da Engenharia de Software que utilizam esses artefatos; no Capítulo 3 são apresentados conceitos relacionados às estruturas de grafos e inerentes ao formalismo de Gramática de Grafos; no Capítulo 4 é descrita a abordagem proposta para extração de gramática de grafos a partir de código-fonte; no Capítulo 5 é apresentado o estudo realizado sobre a metodologia de extração e análise de gramática de grafos a partir Casos de Uso, juntamente com a validação dessa metodologia, através de um estudo empírico, e uma extensão da análise propostas por essa metodologia. Por fim, no Capítulo 6 é feita uma discussão sobre as dificuldades encontradas e alternativas possíveis, e uma análise sobre os resultados obtidos.

1.1 Motivação

Modelos são artefatos muito poderosos no âmbito do desenvolvimento de software, devido a sua aplicabilidade em diferentes subáreas da Engenharia de Software e a variedade de operações que podem ser realizadas sobre os mesmos. Sendo assim, é importante que sejam pesquisadas e exploradas estratégias para manipulação desses artefatos, desde a criação de um modelo, análise de suas características, a definição do tipo de informação que será representada, até a articulação com outras técnicas de desenvolvimento e manutenção de software. Ainda, é fundamental a preocupação com a confiabilidade de um modelo, ou seja, é necessário que a abstração seja fiel ao artefato real representado, seja ele parte da documentação de um software ou a própria implementação. Sendo assim, a utilização de métodos formais para a descrição de um modelo se apresenta como uma alternativa interessante, no sentido de tornar o modelo construído mais confiável e robusto para futuras operações.

A utilização de Métodos Formais proporciona uma segurança muito maior ao desenvolvedor. Porém, uma vez que se deseja utilizar algum desses métodos, existe a necessidade da construção de um modelo formal, ou seja, um modelo baseado em formalismos matemáticos sobre o qual serão realizadas operações. Embora eficazes, nem sempre métodos formais são intensamente utilizados devido, principalmente, ao alto custo computacional envolvido na realização das análises matemáticas. Muitas vezes a utilização de modelos formais ainda é restrita a profissionais que possuem um profundo conhecimento sobre formalismos matemáticos, pois cada método possui suas particularidades, e a familiarização com essas características demanda um investimento de tempo dos desenvolvedores que geralmente é impossibilitado pelo seu ambiente de trabalho. O formalismo de Gramática de Grafos, por sua vez, une as características de um método formal com a expressividade de uma representação gráfica, possibilitando, ao mesmo tempo, a exploração do modelo, através de operações e análises sobre a gramática, e uma compreensão simplificada do sistema, através da visualização gráfica dos componentes e possíveis estados do software modelado.

Dada a extensa aplicação de modelos no desenvolvimento de sistemas computacionais, é autêntica a importância do estudo de suas propriedades, caminhos para construção de modelos e alternativas para a especificação adequada dos mesmos, buscando explorar a contribuição de tais artefatos ao desenvolvimento de software. Além disso, a exploração de abordagens para extração e manipulação de modelos descritos através de técnicas como métodos formais, nesse caso Gramática de Grafos, contribui para que as técnicas e metodologias que operam sobre o formalismo de transformação de grafos prosperem tanto na comunidade acadêmica quanto na indústria de software.

Existem trabalhos que exploram a extração de modelos a partir de código utilizando informações de naturezas diferentes, bem como a extração especificamente para o método de transformação de grafos. No entanto, esses trabalhos, apresentados sucintamente e comentados nos capítulos a seguir, não possuem uma estrutura bem definida ou tem o foco em outros aspectos, que não a extração de modelos em si para a realização de análises, o que resultou na discussão e na proposta de uma nova abordagem, apresentada no Capítulo 4. Da mesma forma, existem trabalhos convergentes para a tradução de diagramas de Caso de Uso para métodos formais, no entanto, possibilitam um pequeno conjunto de análises ou exploram outras possibilidades de manipulação de gramática de grafos, como, por exemplo, a execução do sistema. A metodologia apresentada no Capítulo 5 foi elaborada a partir da necessidade e possibilidade de se realizar um maior número de análises, buscando melhorias na descrição

dos Casos de Uso ao mesmo tempo a criação de um modelo do sistema a partir desses artefatos, o que, nos outros trabalhos, não é o objetivo principal. Assim, busca-se a atuação nos dois extremos do processo de desenvolvimento do software, auxiliando desenvolvedores e *stakeholders* a trabalhar com modelos, descritos através de transformações de grafos, para diversas finalidades.

O esforço dedicado a pesquisas sobre métodos formais tem como objetivo a popularização de formas de utilização de tais métodos, de forma que se tornem acessíveis aos desenvolvedores sem que os mesmos necessitem um extenso conhecimento sobre seus fundamentos matemáticos. Dessa forma, é possível instigar os desenvolvedores a utilizar modelos mais confiáveis para se trabalhar em todas as fases do desenvolvimento. Inicialmente, a criação de representações do sistema com informações mais precisas pode levar à construção de um software mais confiável, isto é, diminuindo a possiblidade de falhas do sistema. Além disso, após a implementação de um software é possível utilizar um modelo formal do mesmo para a realização de análises, tanto para verificar se o objetivo inicial do desenvolvimento foi realmente atingido, quanto para promover melhorias no produto criado, a fim de aumentar a qualidade do software produzido.

1.2 Objetivos

É objetivo principal deste trabalho investigar e analisar abordagens para extração e manipulação de modelos computacionais especificados através do formalismo de gramáticas de grafos, atuando nos dois extremos do processo de desenvolvimento de software, desde a especificação do sistema, previamente à implementação, utilizando artefatos da documentação de software, até a extração de informações diretamente do código-fonte. A criação e difusão de meios para a obtenção de modelos em gramática de grafos antes e depois da implementação de um software, uma ambição direta deste trabalho, torna possível uma diversidade de análises sobre modelos descritos em tal formalismo, como comparações e verificações de comportamento e propriedades relacionadas à correção e validade de software.

Concomitantemente, a utilização do formalismo de transformação de grafos tem como propósito estimular a utilização desse método formal intuitivo, distinto dos demais por não exigir um conhecimento profundo sobre sua fundamentação matemática. Além disso, em virtude de natureza algébrica desse formalismo, é possível realizar análises e operações interessantes sobre modelos especificados em gramática de grafos, possibilitando a exploração de estratégias e técnicas de verificação desses modelos. Por fim, através das abordagens apresentadas e exploradas, busca-se uma aproximação entre as metodologias

investigadas e desenvolvidas na comunidade acadêmica da e a realidade dos processos de desenvolvimento de software das indústrias.

2 MODELOS

O capítulo 2 contém a fundamentação teórica sobre os aspectos da engenharia de software que utilizam ou atuam sobre modelos. A seguir serão apresentadas definições, conceitos e características dos processos inerentes à extração de modelos, além das principais aplicações.

2.1 Definição

Um modelo é caracterizado como uma representação simplificada ou abstrata de um objeto ou ambiente do mundo real, a qual é construída a fim de facilitar a compreensão de tal objeto ou ambiente em relação a algum aspecto específico (GOMAA, 2011). A definição de Modelo e suas características variam de acordo com a área em que o mesmo é utilizado, podendo se referir desde notas em forma textual, equações matemáticas, diagramas, figuras, entre outros.

Na Ciência da Computação um modelo pode ser definido como uma representação que descreve características de um sistema computacional, geralmente contendo informações que reproduzem em sua essência a estrutura e o comportamento do sistema real (WAVEREN ET AL., 2000). Modelos podem ser criados de acordo com a perspectiva que melhor se adaptar ao problema em questão, descrevendo as funcionalidades do software através de alguma notação textual ou gráfica, identificando os componentes do sistema e seus inter-relacionamentos.

É importante observar que a criação de um modelo depende diretamente do contexto onde o objeto modelado está inserido e será analisado, ou seja, quais características desse objeto se deseja observar. Sendo assim, um mesmo objeto pode ser modelado de formas distintas, uma vez que cabe ao modelador a decisão de quais aspectos estarão ou não presentes em um modelo.

2.2 Modelos na Engenharia de Software

De uma forma geral, modelos são utilizado em situações onde deseja-se manipular informações de um sistema, mas, devido sua grandeza ou complexidade, não é necessário ou não é viável a realização de tais operações sobre o sistema como um todo. Modelos de sistemas computacionais são utilizados, de uma forma genérica, em duas situações: análise e síntese. No caso da análise, modelos são usados para identificar e explicar estruturas e/ou

comportamentos de um sistema, enquanto que, no caso de síntese, utiliza-se um modelo para construir um sistema com estruturas e comportamentos desejados (JACOBSEN, 2000).

Modelos são muito utilizados em atividades da Engenharia de Software, devido a sua capacidade de representar sistemas computacionais em níveis de abstração mais altos do que o código-fonte de um software. É possível utilizar modelos tanto em processos da *Engenharia Progressiva*, que engloba as técnicas tradicionais de desenvolvimento de software, quanto na *Engenharia Reversa*, onde são utilizadas técnicas retroativas ao processo de desenvolvimento de software tradicional. Sendo assim, modelos podem ser utilizados desde as primeiras fases de desenvolvimento de um software, como representação dos requisitos e funcionalidades, até mesmo após o final do desenvolvimento, para atividades de análise e facilitando a compreensão do produto final.

2.2.1 Engenharia de Software Progressiva

A *Engenharia Progressiva* ou *Engenharia de Software Tradicional* é a técnica tradicional para o desenvolvimento de software, a qual parte de objetos independentes da implementação e de abstrações e tem como objetivo produzir uma implementação do sistema, seguindo a sequência estabelecida no projeto (OSBORNE, CHIKOFSKY, 1990). O processo de software tradicional é composto por um conjunto de atividades, divididas fundamentalmente nas fases de *especificação}*, *desenvolvimento*, *validação* e *evolução* do software (SOMMERVILLE ET AL., 2008). Essas fases são comuns a alguns modelos de processos de software, no entanto cada modelo de processo pode abordar diferentes necessidades de desenvolvimento, de acordo com o tipo de sistema em questão.

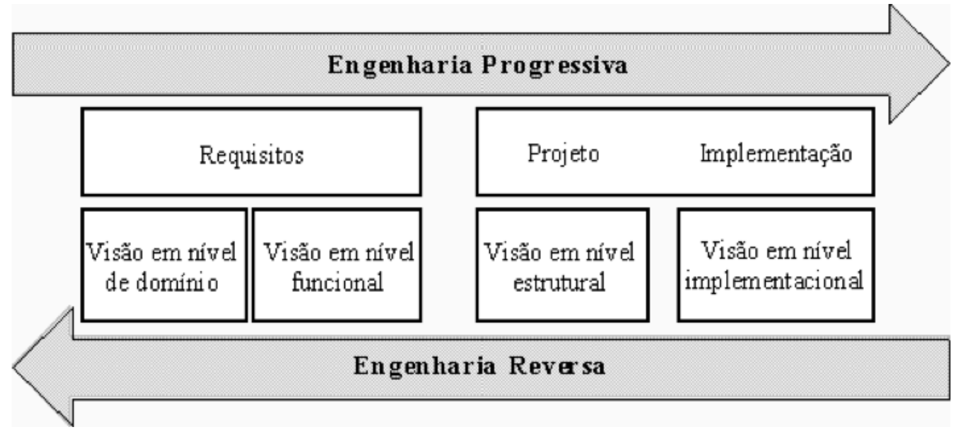
As metodologias de desenvolvimento de software tradicionais foram criadas e desenvolvidas, principalmente, porque a realização de correções ou alterações posteriores à implementação do sistema, na maioria dos casos, eleva significativamente os custos do projeto. Uma vez que o sistema é previamente planejado e especificado, o desenvolvimento é baseado nessas informações, de forma auxiliar o desenvolvedor a construir uma implementação condizente com o proposto no projeto.

A evolução dos computadores promoveu também uma evolução no desenvolvimento de software, pois o contexto de desenvolvimento atualmente é distinto de décadas anteriores. Muitas metodologias novas para o desenvolvimento de software foram desenvolvidas, como as metodologias ágeis, focadas em novas necessidades de software, como diminuição de riscos durante o projeto e aceleração do desenvolvimento, desencadeadas por avanços da tecnologia e exigências do mercado de software. No entanto, é importante fazer a distinção

entre um processo de software e metodologias de desenvolvimento de software. Quando nos referimos a uma metodologia nova, não necessariamente significa que esta metodologia segue um padrão de processo da engenharia de software diferente do tradicional.

Muitas das novas metodologias continuam seguindo o processo de software tradicional, uma vez que o principal aspecto que caracteriza a engenharia progressiva ou engenharia de software tradicional, como mostra a Figura 2.1, é que esse processo parte da especificação escrita do projeto, ou seja, de artefatos representados em um certo nível de abstração, que são utilizados para guiar o desenvolvimento e construir o sistema desejado. Portanto, todas as metodologias, sejam elas recentes ou não, que partem de uma especificação documentada, possuem uma fase de desenvolvimento, passam por uma validação e visam a evolução do software, são caracterizadas como técnicas da engenharia de software tradicional, uma vez que seguem as fases fundamentais desse processo de software.

Figura 2.1 – Ciclo de desenvolvimento de software.



Fonte: Chikofsky, Cross II (1990).

2.2.2 Engenharia de Software Reversa

Engenharia reversa é o processo inverso a engenharia progressiva, caracterizado pelas atividades retroativas ao ciclo de vida de um software, que parte de um baixo nível de abstração para um alto nível (CHIKOFSKY, CROSS II, 1990), conforme apresentado na Figura 2.1. Pressman define a engenharia reversa como um processo de recuperação de projeto, constituído pela análise de um programa e a criação de uma representação do mesmo em um nível de abstração mais alto que o código-fonte (PRESSMAN, 1995).

É possível realizar engenharia reversa de um sistema com ou sem o código-fonte do sistema. Quando não se tem acesso ao código-fonte são utilizados métodos como análise do fluxo de dados ou através tradução da linguagem de máquina por ferramentas como

descompiladores. Por outro lado, quando se tem acesso ao código-fonte podem ser extraídas informações a partir de análises estáticas, como os componentes e as relações que compõem o sistema, ou análises dinâmicas, através de rastros de execução, que consistem no monitoramento do comportamento do programa para determinadas entradas.

A engenharia reversa é intensamente aplicada a situações onde existe um código-fonte legado e é necessário extrair informações desse código para fins de manutenção, por exemplo (YU ET AL., 2005). Uma forma interessante e comumente utilizada para representar as informações extraídas pela engenharia reversa é através de um modelo do sistema, onde é retratado o comportamento e as possíveis ações executadas pelo software. É possível ainda utilizar ferramentas que permitem a realização de operações e análises sobre o modelo.

2.3 Extração de Modelos

O processo de extração de modelos é classificado como uma técnica de engenharia reversa, que por sua vez, inversamente à engenharia progressiva, compreende atividades retroativas ao ciclo de desenvolvimento um software. O objetivo principal da extração de modelos é obter informações de um sistema, criando uma representação diferente ou em um nível de abstração mais alto que o código fonte.

A extração de modelos é definida por (HOLZMANN, SMITH, 1999) como um processo de geração de um modelo fiel do sistema a partir de uma implementação, com o intuito de viabilizar a verificação de propriedades do sistema. Modelos são extraídos com a finalidade de facilitar a realização de alterações em um sistema, possibilitando a utilização de técnicas e ferramentas na verificação de características e propriedades. Através de um modelo de um software é possível analisar se a estrutura e o comportamento do mesmo está de acordo com uma especificação.

Existem diferentes formas de se extrair um modelo de um sistema, que são classificadas de acordo com o tipo de informação que utiliza, podendo ser: baseado em informações estáticas, baseado em informações dinâmicas ou, ainda, híbridas, combinando informações estáticas e dinâmicas de forma complementar.

2.3.1 Modelos baseados em Informações Estáticas

A extração de modelos baseada em informação estática coleta informação diretamente do código fonte ou de código compilado, sem a necessidade da execução do programa. Essa informação pode ser obtida através da análise do fluxo de controle, fluxo de dados ou através de anotações de código que destaquem os pontos interessantes para a análise.

Uma forma de extrair um modelo baseado em informações estáticas é através de um *Control Flow Graph (CFG)* do sistema, que representa, através da notação de grafos, as possíveis execuções de um programa. Os nós inicial e final representam o início e o final da execução do programa, os nós intermediários e as arestas do CFG representam os caminhos possíveis de execução que ligam o nó inicial ao nó final. Ainda, as arestas que somente podem ser acessadas com uma determinada condição, são rotuladas com o predicado que habilita aquela execução. Por meio do CFG, portanto, é possível ter uma noção do que o programa pode ou não fazer.

Ainda, um modelo baseado em informações estáticas representa o sistema de forma que os valores reais das variáveis sejam abstraídas, possibilitando assim a utilização de ferramentas para análise do sistema e validando os resultados para quaisquer valores. Extrair modelos a partir de código-fonte também pode ser classificada como atividade de engenharia reversa e caracterizada como análise estática de um sistema, visando à recuperação de informações sobre os dados ou arquitetura do sistema (SOUZA, 2012).

2.3.2 Modelos baseados em Informações Dinâmicas

Modelos construídos com informações dinâmicas são resultados de um processo de inferência e utilizam informações de execuções de um programa para identificar padrões comportamentais. A informação para a construção desses modelos é coletada por meio de *rastros de execução* (ou *traces*), ou seja, dados da execução do programa gerados para algum conjunto de entradas.

O processo de coleta das informações dos *traces* é feito por meio de um monitoramento da execução do programa. Uma forma de viabilizar esse monitoramento é instrumentar o código-fonte, inserindo anotações em partes do código onde se deseja obter a informação, tornando possível o conhecimento de valores de variáveis do programa, o estado da pilha de execução, ocorrências de chamadas de métodos, entre outras informações em tempo de execução.

A principal vantagem da utilização de informações dinâmicas é que essas informações são referentes a execuções do programa, portanto, comportamentos factíveis, pois, um comportamento que não pode acontecer não aparecerá nos rastros de execução. No entanto, as informações obtidas representam o comportamento do programa para o conjunto de entradas finito sobre o qual o programa operou, ou seja, são casos particulares.

2.3.3 Modelos baseados em Informações Híbridas

A partir das limitações do uso de informações estáticas ou de informações dinâmicas de forma isolada, foi desenvolvida uma abordagem alternativa e interessante para a extração de modelos, que sugere a utilização de ambos os tipos de informações, tendo como resultado um modelo híbrido (DUARTE, 2007). Dessa forma, o conhecimento obtido estática e dinamicamente é usado de forma complementar um ao outro.

Dado que informações estáticas proporcionam um conhecimento geral sobre o comportamento de um programa, o uso de informações dinâmicas pode melhorar a precisão do modelo fornecendo dados sobre comportamentos factíveis específicos. Ao mesmo tempo, enquanto informações dinâmicas fornecem dados sobre comportamentos particulares, as informações estáticas podem tornar possível uma generalização de um comportamento baseado em um conjunto de exemplos.

2.4 Engenharia Dirigida por Modelos

A engenharia dirigida por modelos, ou *Model-Driven Engineering (MDE)*, é uma metodologia que tem como objetivo unificar as iniciativas de melhorias no processo de desenvolvimento de software, por meio da utilização de modelos, tanto na implementação quanto nas etapas de integração, manutenção e testes do sistema de software (DEURSEN, VISSER, WARMER, 2007). Essa abordagem propõe o uso de modelos que capturam características do sistema em um alto nível de abstração, semelhante à Arquitetura Dirigida a Modelos, ou *Model-Driven Architecture (MDA),* a qual utiliza a modelagem como centro do processo de desenvolvimento e parte de um modelo abstrato para o sistema concreto através de transformações de modelos (TRUYEN, 2006).

A abordagem MDE foi idealizada com o intuito de: aumentar a produtividade, maximizando a compatibilidade entre sistemas por meio da reutilização de modelos padronizados; simplificar o processo do projeto, via modelos de padrões de projetos em domínios recorrentes; e promover uma melhor comunicação entre os envolvidos no projeto, padronizando a terminologia utilizada e incentivando a aplicação das melhores práticas no domínio da aplicação. Através de restrições do domínio, podem ser utilizadas ferramentas MDE para realizar verificação de modelos e prevenir erros no início do ciclo de vida do projeto (SCHMIDT, 2006).

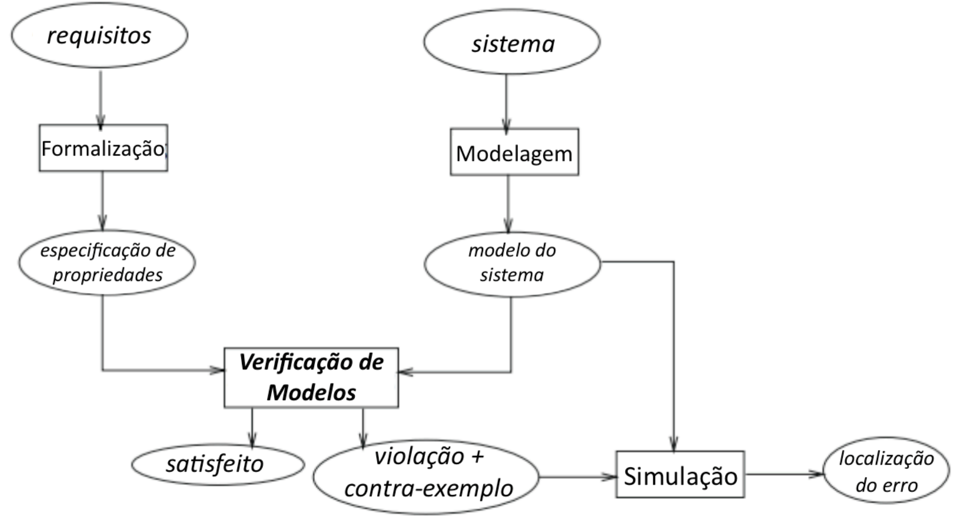
As iniciativas MDE visam a utilização de modelos não somente como uma simples documentação do sistema, mas como uma representação que obedeça determinadas especificação e que possa ser utilizada em ferramentas que executam operações precisas sobre

os modelos (BÉZIVIN, 2005). Na metodologia MDE os modelos são especificados em Linguagens de Domínio Específico, ou *Domain-Specific Language (DSL)*, linguagens dedicadas a um domínio em particular ou a uma técnica de solução particular, com conceitos e noções específicas (DEURSEN, KLINT, VISSER, 2000). As ferramentas MDE utilizam mecanismos de transformação ou de geração que analisam certos aspectos do modelo, com a finalidade de garantir a consistência entre a implementação e a especificação dos requisitos do sistema, viabilizando a criação de um sistema *correto-por-construção*.

2.5 Verificação de Modelos

A Verificação de Modelos ou *Model Checking* (CLARKE JR., GRUMBERG, PELED, 1999) consiste em uma técnica para automatizar a verificação de sistemas representados por modelos descritos em uma linguagem de especificação (WING, 1990), geralmente um formalismo de estados finitos. Essa técnica é composta por três fases: *modelagem*, *especificação* e *verificação*; e seu processo completo é representado graficamente na Figura 2.2.

Figura 2.2 – Visão geral da técnica de Verificação de Modelos (Model Checking).



Fonte: Baier and Katoen (2008).

A modelagem consiste na geração de um modelo aceito por alguma ferramenta responsável por executar a verificação, denominada *Model Checker*; a especificação é a definição de um conjunto de propriedades que o sistema deve satisfazer em uma lógica temporal; e a verificação explora todo o espaço de estados do modelo, conferindo se as propriedades especificadas são satisfeitas em todos os estados do sistema e, caso as

propriedades não sejam atendidas, são gerados contra-exemplos que mostram as situações onde tal propriedade não é válida.

A abordagem utilizada na Verificação de Modelos possibilita que a verificação seja executada de forma automática, sem a necessidade de um auto nível de interação com o usuário ou conhecimento avançado da técnica. É possível também realizar uma verificação parcial do modelo, ou seja, as propriedades podem ser verificadas individualmente, sem a necessidade de uma especificação completa dos requisitos. Além disso, a verificação de modelos não é vulnerável a erros, uma vez que explora todas as possíveis execuções do modelo, garantindo a correção completa do modelo de acordo com uma determinada especificação (BAIER, KATOEN, 2008).

O principal desafio da técnica de Verificação de Modelos é tratar a explosão do espaço de estados pois, por utilizar uma abordagem de força bruta, facilmente pode ser atingindo um número de estados que exceda a capacidade da memória. A restrição no tamanho do modelo exige que, na etapa de modelagem, haja um esforço para se obter um modelo do sistema finito e com o menor tamanho possível. Sendo assim, é possível observar a importância do processo de construção de um modelo, fundamental na verificação de modelos e em outras técnicas de Engenharia de Software, onde sua eficiência pode ser fator decisivo para o sucesso da realização de atividades de análise sobre um determinado sistema.

3 GRAMÁTICA DE GRAFOS

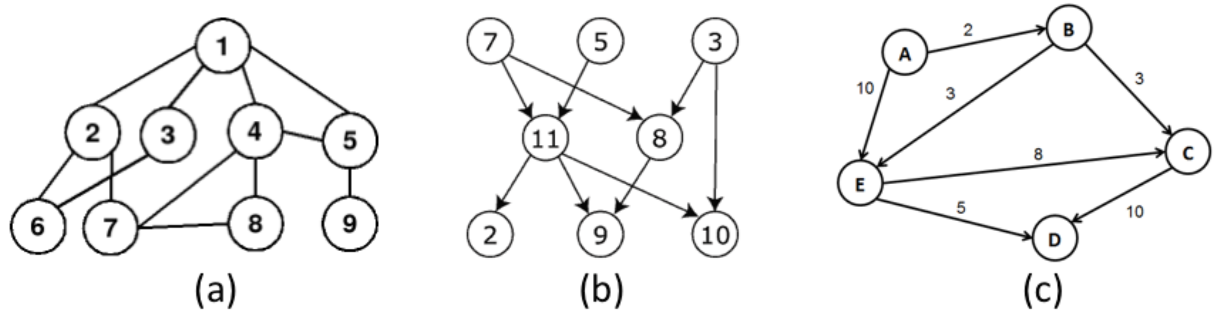
O Capítulo 3 apresenta uma introdução sobre Grafos e um referencial teórico sobre o formalismo de Gramática de Grafos. A seguir serão apresentados características da modelagem de sistemas através de grafos e definições e conceitos relacionados ao formalismo de transformações de grafos.

3.1 Grafo

Grafo é uma abstração que permite a codificação de relacionamentos entre objetos. Um grafo é composto por um conjunto de vértices (*V*), que representam elementos, e arestas (*E*), que interligam esses elementos. Grafos possuem uma semântica bem definida, uma vez que são estruturas algébricas, e podem ser utilizado para explicar situações complexas de forma compacta, visual e intuitiva (BONDY, MURTY, 2007).

As arestas dos grafos podem ser direcionadas, nesse caso os grafos são chamados de *Dígrafos* e suas arestas são definidas por pares ordenados de vértices, representando a origem e o destino. Ainda, é possível atribuir valores algébricos ou rótulos para os vértices e as arestas, chamados de atributos. A Figura 3.1 mostra exemplos de um grafo simples (a), um grafo direcionado (b) e um grafo no qual as arestas possuem um rótulo, no caso específico um peso associado.

Figura 3.1 – Exemplos de Grafos: (a) grafo simples não direcionado; (b) dígrafo ou grafo direcionado; (c) grafo com pesos associados às arestas.



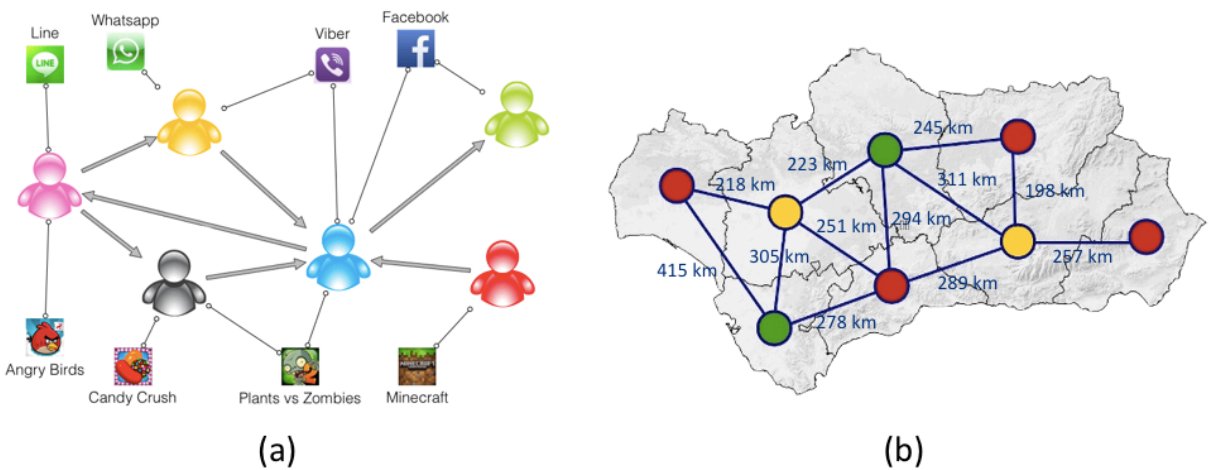
Fonte: (do autor , 2016).

Na área da Ciência da Computação, grafos são amplamente utilizados na modelagem de problemas (EHRIG, 1999), por possuírem uma representação gráfica intuitiva, facilitando a interpretação do modelo, e por existirem algoritmos para solução de problemas com grafos

bastante difundidos, que exploram a manipulação e o percurso entre os diferentes caminhos possíveis em um grafo.

Entre as principais aplicações do formalismo de grafos na especificação e modelagem de problemas na computação, destacam-se principalmente a identificação de relacionamentos entre elementos interligados em redes e a exploração e análise sobre caminhos possíveis entre diferentes arestas de um grafo, como mostra a Figura 3.2. No lado esquerdo *(a)*, são representados os relacionamentos entre dois tipos de elementos, pessoas e softwares para dispositivos móveis, onde as arestas representam a utilização de um aplicativo por um usuário e o relacionamento entre usuários. Esse tipo de grafo é frequentemente utilizado para a identificação de padrões comportamentais entre usuários desses aplicativos. No grafo do lado direito *(b)* da Figura 3.2, estão representadas as conexões e distâncias entre cidades em um mapa geográfico, onde podem ser analisadas diferentes formas de se percorrer um determinado trajeto entre dois pontos.

Figura 3.2 – Exemplos da aplicação de grafos na especificação de relacionamentos: (a) análise comportamental de usuários de aplicativos para dispositivos móveis; (b) representação de adjacências e distâncias entre territórios.



Fonte: (do autor , 2016).

3.2 Definição de Gramática de Grafos

Gramática de Grafos (GG) ou *Sistema de Transformação de Grafos* é um poderoso formalismo, amplamente conhecido para a especificação de sistemas, destacando-se dos demais métodos formais principalmente por ser um método intuitivo, possibilitando, de forma simples, a representação de sistemas concorrentes e distribuídos (EHRIG, 1997). A especificação de um sistema por meio desse formalismo é composta por grafos, que

representam os estados e a configuração do sistema, e regras, que são aplicadas aos grafos e representam transições entre estados do sistema, ou seja, expressam os comportamentos possíveis do sistema.

A ideia básica do formalismo é que o sistema seja representado naturalmente como um grafo, de acordo com um certo nível de abstração, e as transformações de estados possam ser expressas por processos de reescrita dos grafos (EHRIG, 1979). A evolução do sistema é determinada pela aplicação de regras, a partir de um grafo inicial, guiados por um grafo tipo, o qual determina o tipo dos elementos que podem estar presentes nos grafos. A principal vantagem do formalismo de transformações de grafos é que através dele é possível criar um modelo visual e intuitivo do sistema provido de uma semântica precisa.

O método de Gramática de Grafo é uma generalização das Gramáticas de Chomsky, onde são substituídas as *strings* por grafos (ROZENBERG, 1997). Esse formalismo possui também relação com o modelo de Redes de Petri, pois uma gramática de grafos pode também ser considerada como uma generalização de Redes de Petri (KORFF, RIBEIRO, 1996), (CORRADINI, 1995), uma vez que permite mudanças dinâmicas na topologia do sistema e referências entre *tokens*.

3.2.1 Sintaxe de Gramática de Grafos

Uma Gramática de Grafos é definida basicamente por três elementos: um *grafo inicial*, que representa o estado inicial do sistema; um *grafo tipo*, uma representação do conjunto de elementos e arestas permitidos no sistema; e um *conjunto de regras*, composto por todas as possíveis transições entre estados do sistema. A seguir serão detalhados os elementos de uma gramática de grafos e será utilizado como exemplo um sistema computacional que realiza tarefas de acordo com a arquitetura cliente/servidor, onde mensagens com dados são trocadas entre os objetos do sistema, apresentado em Machado (2012).

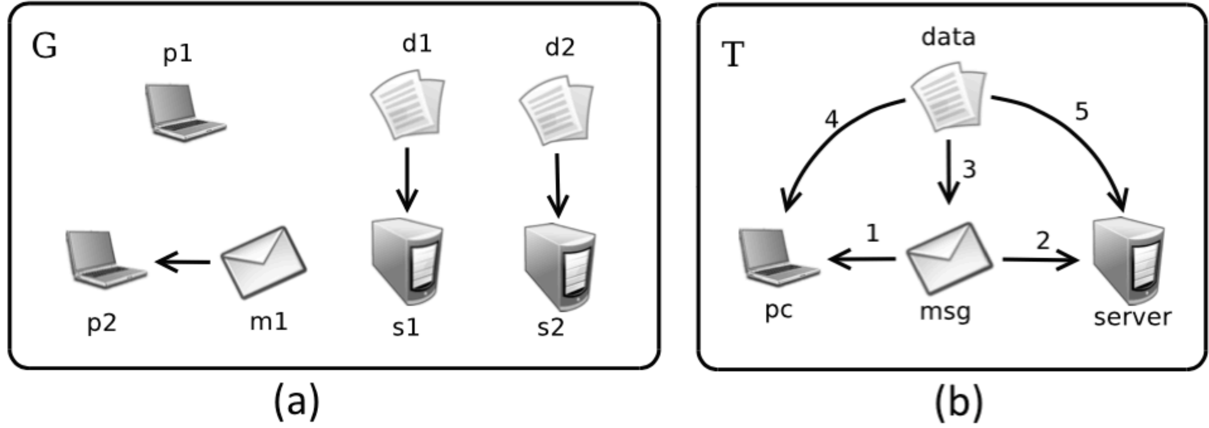
3.2.1.1 Grafo Inicial

O *grafo inicial* consiste em uma representação do estado inicial do sistema, através do formalismo de grafos. As características desse grafo correspondem à configuração do sistema no ponto de inicio da sua execução, bem como os elementos necessários para que o sistema possa ser iniciado. A Figura 3.3 *(a)* apresenta um exemplo de um grafo *G*, que poderia representar o estado inicial de um sistema.

3.2.1.2 Grafo-tipo

É possível utilizar mecanismos de tipagem nos grafos de uma gramática, para isso utilizamos rótulos para os vértices e/ou arestas, e atributos, que podem receber valores de acordo com os tipos de dados utilizados. Através de um *grafo-tipo* é possível especificar quais os tipos de arestas e vértices podem existir na gramática, definindo os rótulos, que identificarão um elemento, e os atributos a serem utilizados, como apresentado na Figura 3.3 *(b)*, que poderão representar, por exemplo, propriedades de um sistema. Os mecanismos de tipagem colaboram para que a especificação de um sistema se torne mais compacta, simples e facilita a compreensão do mesmo por parte do usuário. Nesse exemplo, através de uma análise do grafo-tipo *T* é possível observar que o sistema possui quatro tipos de elementos, que são os vértices de *T* (*data*, *msg, pc e server*) e representam 4 elementos do sistema (dados, mensagens, computadores e servidores), e cinco tipos de arestas, as quais representam os relacionamentos entre os objetos do sistema, que, nesse caso, se definem como ligações entre os objetos, ou seja, uma aresta a do vértice v_1 para v_2 indica que v_1 está ligado a v_2.

Figura 3.3 – Exemplo de elementos da sintaxe de uma Gramática de Grafos: (a) um grafo *G* tipado, representando um estado sistema, que pode ser inicial ou não, construído de acordo com um grafo-tipo; (b) um grafo-tipo do sistema que representa todos os tipos de arestas e vértices existentes no sistema.



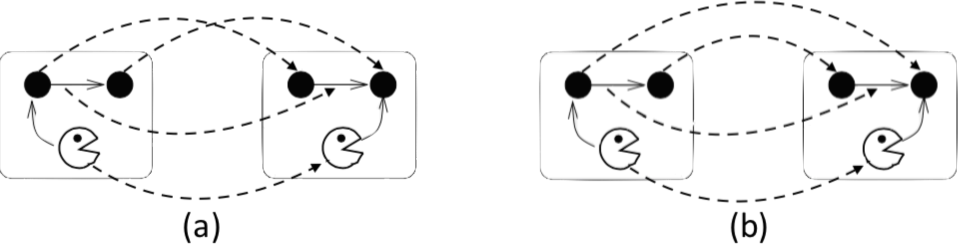
Fonte: Machado (2012).

3.2.1.3 Regras

Uma *Regra* de grafos *r : L→R* é composta por dois grafos, o lado esquerdo (*L*) e o lado direito (*R*), e um homomorfismo parcial de grafos *r*, o qual mapeia vértices e arcos de *L* em vértices e arcos de *R*. A Figura 3.4 ilustra essa situação, utilizando o exemplo do deslocamento do personagem do jogo *Pac-man* entre duas posições de um possível mapa,

representadas por um vértice com o desenho de um círculo preenchido. Dessa forma, cada vez que um arco e_L for mapeado para um e_R o vértice de origem de e_L deve ser mapeado para o vértice de origem e_R, analogamente para o vértice destino, caso contrário não há um morfismo de grafos.

Figura 3.4 – (a) morfismo de grafos; (b) não é morfismo de grafos.



Fonte: Ribeiro (2000).

A aplicação de uma regra *r : L→R* consiste na identificação do subgrafo *L* correspondente ao lado esquerdo da regra no grafo do sistema e, substituição do subgrafo *L* pelo subgrafo *R*, ou seja, pelo lado direito a regra. Esse processo caracteriza a passagem de um estado do sistema para outro, e ocorre da seguinte forma:

- Os itens em *L* que não possuem imagem em *R* são deletados;
- Os itens em *L* que são mapeados para *R* são preservados;
- Os itens em *R* que não tem uma pré-imagem em *L* são criados.

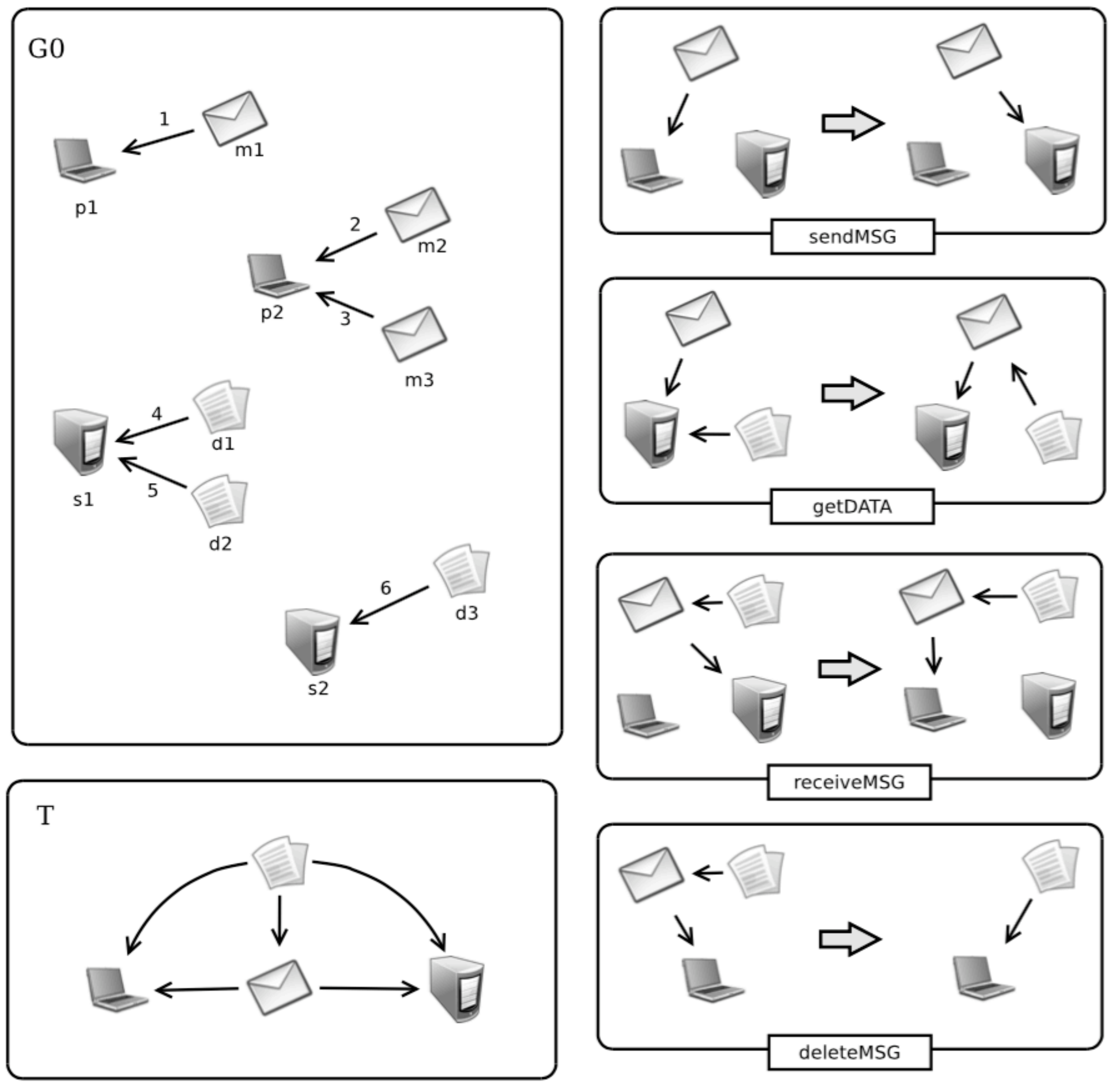
3.2.1.4 Exemplo de Gramática de Grafo

A Figura 3.5 apresenta um exemplo de Gramática de Grafos, através da definição dos três elementos de sintaxe, o grafo inicial (*G0*), o grafo-tipo (*T*) e o conjunto de regras (*sendMSG*, *getDATA*, *receiveMSG* e *deleteMSG*). Essa gramática representa um sistema computacional, ou seja, um modelo do sistema em um certo nível de abstração (grafos) formalmente definido através de regras (gramática de grafos), que pode ser utilizado como entrada, com diferentes propósitos, em atividades de engenharia de software.

O sistema representado pela Figura 3.5 simula um ambiente cliente/servidor, onde há troca de mensagens e dados entre computadores e servidores. Cada regra do conjunto de regras representa uma ação: a regra *sendMSG* representa o envio de uma mensagem por um cliente (computador) a um servidor; a regra *getData* representa o preenchimento de uma mensagem, pelo servidor, com os dados solicitados; a regra *receiveMSG* representa o recebimento de uma mensagem do servidor, pelo cliente; e a regra *deleteMSG* representa a

transferência de dados de uma mensagem para um cliente, bem como a exclusão dessa mensagem ao final da ação.

Figura 3.5 – Exemplo de uma gramática de grafos para um sistema cliente/servidor: grafo inicial (*G0*), grafo-tipo (*T*) e o conjunto de regras (*sendMSG*, *getDATA*, *receiveMSG* e *deleteMSG).*



Fonte: Machado (2012).

É importante observar que, em alguns casos, mais de uma regra podem ser aplicadas a um grafo do sistema, ou seja, existe uma determinada configuração do sistema que habilita a aplicação de duas ou mais regras. Se as regras alteram elementos diferentes no grafo do sistema elas podem ser aplicadas paralelamente, porém, nos casos em que duas ou mais regras realizam uma operação sobre um mesmo item, por exemplo, uma das regras apaga um nó do

grafo que é consumido por outra regra, a aplicação dessa primeira regra impossibilita a aplicação da segunda, o que é chamado de *conflito*. Quando essa situação ocorre o sistema escolhe de forma não determinística qual das regras será aplicada.

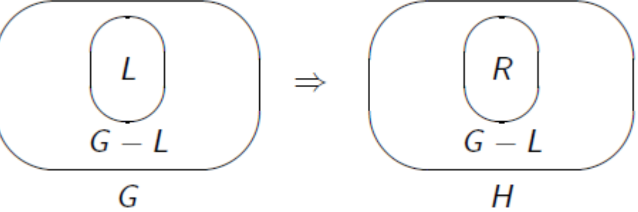
3.2.2 Semântica de Gramática de Grafos

A semântica da gramática de grafos está relacionada com o comportamento do sistema modelado pelo formalismo. Os comportamentos possíveis de um sistema, em gramática de grafos, são definidos pelas regras e as mudanças no estado do sistema ocorrem devido a um processo chamado *passo de derivação* que acontece quando uma regra $r : L \rightarrow R$ é aplicada a um grafo G.

3.2.2.1 Passo de Derivação

Um *passo de derivação* somente é possível se existe um *match* entre uma regra e um grafo do sistema. Um *match* acontece quando há uma ocorrência do lado esquerdo L de uma regra $r : L \rightarrow R$ em um grafo G. O subgrafo L é então substituído pelo subgrafo R, lado direito da regra r, como ilustra a Figura 3.6. É importante observar que o grafo G pode ainda conter outros elementos, os quais, se não fazem parte do subgrafo correspondente ao lado esquerdo L da regra em questão, não sofrem alteração.

Figura 3.6 – Exemplo de *Passo de Derivação*: Aplicação de uma regra $r : L \rightarrow R$ a um grafo G.



Fonte: (do autor , 2016).

O grafo G representa o estado do sistema antes do passo de derivação, enquanto que o grafo H caracteriza o sistema após o passo de derivação. No grafo G é identificado o subgrafo L, habilitando, portanto, a aplicação da regra $r : L \rightarrow R$. A aplicação da regra consiste em substituir, no grafo G, o subgrafo L pelo subgrafo R, criando um novo grafo H, composto pelos elementos de G que não sofrem alteração e pelo subgrafo R.

3.3 Regras Concorrentes

Uma *Regra Concorrente (RC)* é caracterizada pela sua capacidade de representar a aplicação de uma *Sequência de Regras (SR)*. Dado um conjunto de regras e sua ordem sequencial de aplicação, é possível construir uma única regra que possui o efeito de todas essas regras aplicadas na ordem estabelecida. Uma *RC* é capaz de sintetizar o comportamento do sistema quando uma *SR* é aplicada.

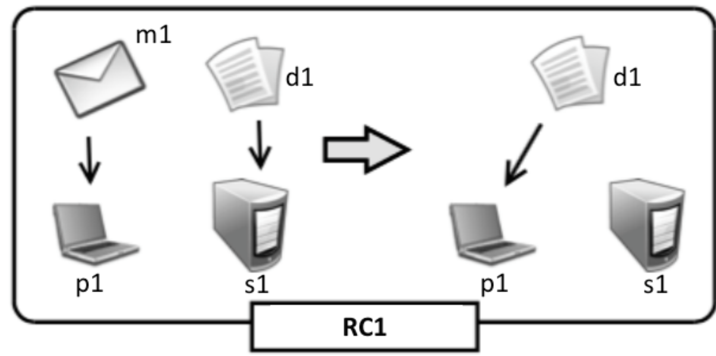
Existem diferentes formas de se construir uma *RC* (RUNGE, ERMEL, TAENTZER, 2012), explorando relações de dependências entre as regras, paralelismo ou predefinindo um fluxo entre as regras. Existem, também, ferramentas como o *The Attributed Graph Grammar System (AGG)* (TAENTZER, 2000), que possibilitam a construção construir *RCs* de forma automática.

Sobre a estrutura da *RC*, o seu lado esquerdo representa o estado necessário do sistema para habilitar a aplicação da *SR* correspondente, contendo todos os itens utilizados pelas regras dessa sequência. Isso porque, dentre as regras pertencentes à *SR* em questão, se uma regra consome um item que não é criado por nenhuma das regras anteriores a ela, esse item deve existir no sistema antes da aplicação da primeira regra da sequência, ou seja, precisa aparecer no lado esquerdo da *RC*. Sendo assim, os itens que aparecem no lado esquerdo da *RC* e não aparecem no lado direito da mesma foram consumidos por alguma regra da sequência. Analogamente, o lado direito da *RC* representa a configuração do sistema após a aplicação da *SR*. Logo, todos os elementos que aparecem no lado direito da *RC* e que não aparecem no seu lado esquerdo foram criados por alguma regra da sequência e não foram consumidos por nenhuma outra.

Utilizando como exemplo a Gramática de Grafos apresentada na Figura 3.5 e supondo uma Sequência de Regras, SR_1, contendo as regras *sendMSG*, *getDATA*, *receiveMSG* e *deleteMSG*, nesta ordem, é possível construir uma regra concorrente, RC_1, gerada a partir da SR_1, a qual é apresentada na Figura 3.7. É, portanto, possível observar que no lado esquerdo de RC_1 aparecem todos os elementos necessários para a aplicação das regras de SR_1, por exemplo, o item do tipo *dados*, denominado d_1, necessário para a aplicação da regra *getDATA* (segunda regra da SR_1) não é criado pela primeira regra da SR_1, portanto, para a aplicação da sequência SR_1 ser possível, o lado esquerdo da regra concorrente RC_1 precisa ter pelo menos um objeto do tipo *dados*, sobre o qual será aplicada a regra *getDATA*. Ainda sobre a estrutura da regra concorrente, no seu lado direito tem-se o grafo resultante, após a aplicação de todas as regras de SR_1, ou seja, o estado do sistema após a aplicação da sequência de regras. É importante observar que a *RC* apresentada na Figura 3.7 é a RC com máxima dependências

entre os elementos das regras da sequência. Demais possibilidades de geração de *RCs* serão apresentadas e exploradas nos Capítulos seguintes.

Figura 3.7 – Exemplo de uma regra concorrente gerada a partir da Sequência de Regras *SR1*.



Fonte: Machado (2012).

3.4 Pares Críticos

A análise de *Pares Críticos* em gramáticas de grafos está diretamente relacionada aos conflitos e dependências existentes entre as regras de uma gramática. Quando duas regras não possuem conflitos ou dependências entre si, significa que as mesmas podem ser aplicadas ao grafo simultaneamente. No entanto, quando duas regras *r1* e *r2* possuem alguma relação entre si, seja de conflito ou dependência, a aplicação de uma das duas regras pode inviabilizar a aplicação da outra regra. Nessas situações, para compreender o comportamento do sistema é necessário analisar a semântica dos relacionamento de conflito e dependência entre as regras.

Um par crítico existe entre duas regras que podem ser aplicadas a um mesmo grafo *G*, quando a aplicação de uma delas inviabilizar a aplicação da outra. O par crítico pode caracterizar um conflito entre duas regras, em uma situação onde apenas uma das duas pode ser aplicada, ou ainda, pode indicar uma dependência entre duas regras, quando as mesmas precisam ser aplicadas em uma determinada ordem. É importante observar que a análise de pares críticos utiliza informações estáticas, ou seja, é independente da execução do sistema e do grafo inicial, uma vez que somente é utilizado o conjunto de regras da gramática para essa análise.

3.4.1 Análise de Conflitos

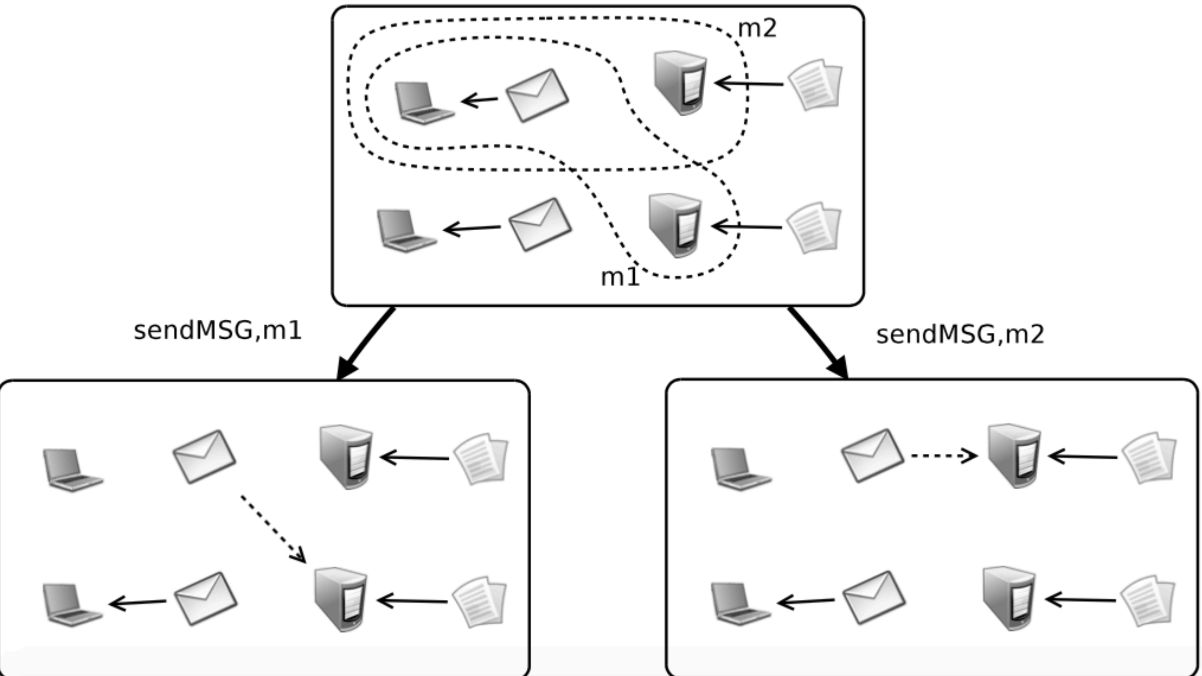
Um *conflito* entre duas regras de gramáticas de grafos existe quando a aplicação de uma das duas regras impossibilita a aplicação da outra. Dadas duas regras *r1* e *r2*, que ao

serem aplicadas a um grafo *G* alteram o estado do sistema para duas configurações distintas G_1 e G_2, respectivamente, existe um conflito entre essas duas regras quando a regra *r1* não puder ser aplicada sobre o grafo G_2 ou a regra *r2* não puder ser aplicada ao grafo G_1.

Um par crítico de conflito, de maneira geral, ocorre quando a primeira regra consome um elemento do grafo que é requerido pela segunda, por exemplo, se *r1* deleta um item do grafo que também é consumido por *r2*. Logo, em um grafo *G*, somente uma entre duas regras conflitantes pode ser aplicada, pois a aplicação de uma desabilita a aplicação da outra. A Figura 3.8 apresenta um exemplo de conflito, onde nos dois casos a mesma aresta é deletada, provocando, então, um conflito.

Conflitos acontecem em três diferentes situações: na primeira, ocorre um conflito quando uma regra deleta um objeto do grafo que a outra regra consome, impossibilitando a aplicação da segunda; na segunda situação, a aplicação de uma regra gera objetos do grafo que não podem existir devido a alguma condição de aplicação negativa (define o que não pode existir no grafo, para que uma regra seja aplicada); e a terceira, quando uma regra altera atributos do grafo que impossibilitem a aplicação de outra regra. É importante observar que as duas primeiras estão relacionadas à estrutura do grafo, enquanto a última é relacionada aos atributos.

Figura 3.8 – Exemplo de conflito no sistema cliente/servidor.



Fonte: Machado (2012).

Semanticamente, um conflito pode indicar comportamentos diferentes de um sistema. Inicialmente, no caso apresentado acima, o grafo do sistema possui duas mensagens e dois servidores capazes de receber uma mensagem e em cada situação essa mensagem é enviada para um deles. Um conflito pode ainda ser identificado como um ponto de decisão no sistema, onde somente um ou outro caminho na execução pode ser seguido, ou uma ação que pode ser inúmeras vezes repetida, quando existe um auto-conflito de uma regra, ou seja, existe um conflito de uma regra com ela mesma. Dessa forma, temos um estado do sistema com duas possibilidades, mas sobre o qual somente uma regra é realmente aplicada. Quando essa situação acontece, a escolha de qual regra será aplicada é não determinística.

É importante observar que uma regra *r1* pode desabilitar a aplicação de uma regra *r2*, mas não necessariamente o contrário, por isso o conflito possui uma natureza assimétrica. A computação dos pares críticos existentes em uma gramática de grafos é feita através da análise de cada regra em relação a si própria e às demais, produzindo, para um conjunto de n regras, uma *matriz de conflitos* de dimensões $n \ x \ n$, onde cada dentro de célula está representado o número de elementos que estão em conflito entre a regra da linha e a regra da coluna.

A Figura 3.9 apresenta a matriz de conflitos para a gramática de grafos utilizada como exemplo até agora, a qual representa um sistema cliente/servidor. A geração da matriz de conflitos não é uma tarefa trivial, pois o número de comparações necessárias é elevado, no entanto, existem ferramentas que realizam a identificação de conflitos de forma automática, como a ferramenta AGG, a qual foi utilizada para a geração da matriz utilizada como exemplo.

Figura 3.9 – Exemplo de matriz de conflitos entre as regras do sistema cliente/servidor gerada através da ferramenta AGG.

Minimal Conflicts

Show

first \ sec...	1	2	3	4
1 sendMSG	2	0	0	1
2 getDATA	0	3	0	0
3 receiveMSG	0	2	6	0
4 deleteMSG	1	0	0	1

Fonte: (do autor , 2016).

3.4.2 Análise de Dependências

Através da análise de pares críticos em Gramáticas de Grafos também podemos identificar relacionamentos de dependência entre regras. Uma dependência entre duas regras existe quando uma delas somente pode ser aplicada após a outra, isto é, dadas duas regras r_i e r_j, existe uma dependência entre essas duas regras quando a regra r_j somente puder ser aplicada a um grafo G após a aplicação da regra r_i sobre o mesmo grafo G.

Existem três situações que caracterizam uma dependência entre regras: a primeira ocorre quando um elemento é criado pela primeira regra e deletado pela segunda; outra existe quando a segunda regra preserva um elemento criado pela primeira regra; e a última acontece nos casos em que a primeira regra preserva um elemento que é deletado pela segunda. As dependências entre regras tem papel importante na reescrita de grafos, pois podem impossibilitar a execução paralela e impôr uma determinada ordem de execução.

Analisando o comportamento de um sistema, através da análise das dependências podemos identificar situações onde para se atingir determinado ponto do sistema, obrigatoriamente, foi necessário passar por um estado anterior. Supondo que em um sistema, para um usuário realizar uma operação O, obrigatoriamente, o mesmo deve ter, previamente, realizado uma operação de autenticação A, por exemplo. A regra que representa a operação O que o usuário deseja realizar é dependente da regra que representa a autenticação A, logo, existe um par crítico de dependência entre essas duas regras.

Figura 3.10 – Exemplo de dependência no sistema cliente/servidor.

Fonte: Machado (2012).

No exemplo apresentado na Figura 3.10 é possível observar que as regras *getDATA* e *receiveMSG* formam um par crítico de dependência, pois para a regra *receiveMSG* ser aplicada sobre o grafo do sistema é necessário que, previamente, a regra *getDATA* altere a configuração do sistema, conectando um elemento do tipo *data* à mensagem, isto é, criando uma aresta entre esses dois elementos. Dessa forma, a aplicação da regra *getDATA* habilita a aplicação da regra *receiveMSG*, caracterizando uma dependência do tipo *criação-leitura*.

Da mesma forma que os conflitos, as dependências também possuem uma natureza assimétrica e a *matriz de dependências* é produzida através da análise de cada regra em relação a si própria e às demais. A diferença é que dentro de cada célula da matriz de dependências está representado o número elementos que geram a dependência entre a regra da linha e a regra da coluna. A Figura 3.11 exibe a matriz de dependências produzidas para o exemplo do sistema cliente/servidor, utilizado como exemplo, através da ferramenta *AGG*.

Figura 3.11 – Exemplo de matriz de dependências entres as regras do sistema cliente/servidor gerada através da ferramenta AGG.

Minimal Dependencies

Show

first \ sec...	1	2	3	4
1 sendMSG	0	1	2	0
2 getDATA	0	0	3	0
3 receiveMSG	2	0	0	1
4 deleteMSG	0	0	0	0

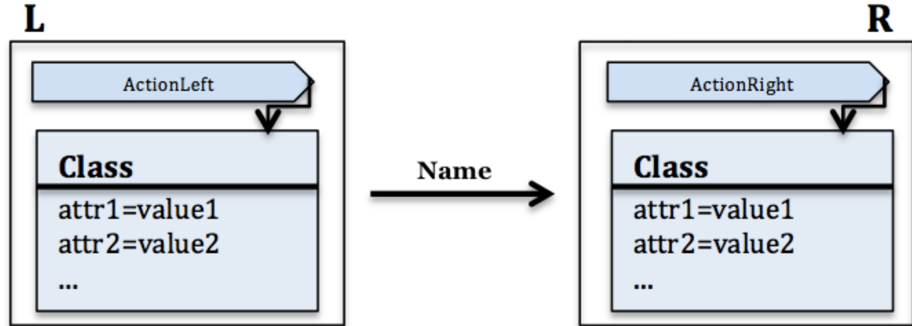
Fonte: (do autor , 2016).

4 EXTRAÇÃO DE GRAMÁTICA DE GRAFOS A PARTIR DE CÓDIGO

Neste capítulo são apresentadas estratégias para a extração de modelos a partir de artefatos de um projeto de software, utilizando o formalismo de transformações de grafos, juntamente com metodologias para análises desses modelos. Em vista das adversidades discutidas previamente, é proposta uma abordagem para extração de modelos descritos através de gramática de grafos, obtido automaticamente a partir das informações de execução de um código Java. Através de uma ferramenta, é extraído um modelo de comportamento do programa, o qual combina informações de fluxo de controle e valores de atributos na forma de rastros de execução. Esta combinação de dados estáticos (fluxo de controle) e dados dinâmicos (valores de atributos) baseia-se na ideia de contextos (DUARTE, KRAMER, UCHITEL, 2006). A partir de uma tabela de contextos e de rastros de contextos, o modelo é especificado formalmente através de uma gramática de grafos.

O formalismo gramática de grafos, detalhado no Capítulo 3, é utilizado por ser um método formal intuitivo para a especificação de sistemas, que possibilita, de forma simples, a representação de aspectos como concorrência e distribuição de sistemas. Em gramática de grafos os estados do sistema são representados por grafos e as transições são representadas por regras, caracterizando-se como uma especificação formal e ao mesmo tempo fornecendo uma representação visual do sistema, ou seja, tem-se uma sintaxe formal e semântica, sem ambiguidades e uma especificação que pode ser utilizada para provar propriedades desse sistema.

Figura 4.1 – Modelo de regra da gramática de grafos utilizada.



Fonte: (do autor , 2016).

A Figura 4.1 apresenta o modelo de regra da gramática de grafos utilizado, onde tanto o lado esquerdo (L) quanto o direito (R) representam contextos do programa, compostos por dois elementos: a classe da aplicação e ações que podem ocorrer nesses contextos.

4.1 Metodologia

Inicialmente, para viabilizar a geração de uma gramática de grafos, o código-fonte de uma aplicação é instrumentado, adicionando-se anotações que representam informações relevantes sobre o comportamento do sistema, como chamadas de métodos e entradas e saídas em estruturas de controle e seleção. Após a execução do código instrumentado, um modelo híbrido do programa é construído, baseado em contextos, responsáveis por conter informações de fluxo de controle e valores de atributos da aplicação. A Figura 4.2 apresenta uma visão geral do processo, indicando todas as etapas realizadas, desde o código-fonte original até a geração da gramática de grafos.

Figura 4.2 – Fluxograma do processo realizado.

Instrumentar código-fonte original → Executar código-fonte instrumentado → Extrair rastros de execução e tabela de contextos → Executar algoritmo de geração da gramática de grafos → Realizar análises sobre a gramática de grafos gerada

Extração do Modelo

Geração da Gramática de Grafos

Fonte: (do autor , 2016).

As primeiras três etapas do processo podem ser classificadas como estágios da extração de um modelo do sistema, incluindo instrumentação do código-fonte, execução do código-fonte instrumentado e a extração de informações dos rastros de contextos. Para instrumentar o código-fonte Java é utilizada a linguagem TXL (CORDY ET AL., 2002), adicionando-se anotações ao código de acordo com algumas regras especificadas previamente. Na segunda fase, o código instrumentado é executado a fim de gerar rastros de execução, armazenados em arquivos de registros. Na terceira etapa, a informação contida nos *Arquivos de Registros de Execução*, denominados alternativamente como *log files*, é usada para identificar contextos. Os contextos são armazenados em uma *Tabela de Contextos (TC)* e as sequências com alternâncias entre contextos e ações formam nossos *Rastros de Contextos (RC)*. As etapas restantes do processo estão relacionadas com a geração e análise da gramática de grafos. No quarto passo, usamos as informações sobre os contextos, TC e RC, para gerar uma gramática de grafos através de um algoritmo, que mapeia informação de contextos para regras de uma gramática de grafos. Finalmente, através de ferramentas de manipulação de

gramática de grafos, realizamos análises sobre a gramática de grafos gerada, possibilitando a verificação de propriedades desejadas. Os arquivos utilizados para a geração da gramática são obtidos através dos procedimentos descritos, em detalhes, em Duarte (2007), e foram extraídos a partir de aplicações que serão exploradas a seguir.

4.1.1 Extração do Modelo

O objetivo dos primeiros três passos do processo descrito acima é obter um modelo comportamental do programa, utilizando o conceito de *contextos* (DUARTE, 2007). Contextos representam uma combinação de um certo ponto do fluxo de controle do sistema juntamente com os valores dos atributos do sistema naquele ponto, definindo assim um estado abstrato do sistema. O modelo é gerado utilizando tanto informações estáticas quanto dinâmicas, armazenando essas informações na tabela de contextos e nos rastros de execução, respectivamente, sendo assim classificado como um modelo híbrido.

Extraímos, então, de um código Java anotado para coletar informações de contexto, uma tabela de contextos, apresentada na Figura 4.3, e um conjunto de rastros de execução, detalhados na Figura 4.5, os quais são utilizados, na metodologia descrita em (DUARTE, 2007), para gerar modelos de sistemas de transição rotulados (LTS). Essa extração é realizada através da ferramenta *Labelled Transition System Extractor* (LTSE) (DUARTE, KRAMER, UCHITEL, 2008). A abordagem aqui proposta utiliza os dados extraídos pela ferramenta para gerar uma gramática de grafos com regras que representam as possíveis transições do sistema.

A aplicação utilizada como exemplo é chamada *Traffic Lights*, e será detalhada a seguir. Essa aplicação simula um controlador de um semáforo, com duas luzes (verde e vermelha) onde somente uma pode estar ligada por vez. Essa aplicação possui chamadas a métodos que podem alterar os atributos do sistema. A gramática dessa aplicação é representada por vértices que representam os contextos e as mensagens.

4.1.1.1 Tabela de Contextos

A tabela de contextos apresenta uma identificação numérica unívoca do contexto identificado durante a execução (coluna *S*), o predicado correspondente ao bloco de comando do fluxo de controle que gerou o contexto (coluna *P*), um identificador do bloco de comando executado (coluna *ID*), o valor de avaliação do predicado naquele ponto da execução (coluna *V*), valores de atributos considerados no contexto (coluna *A*) e um controle da pilha de chamadas ativas (última coluna da tabela). Assim, por exemplo, o contexto *3* é definido por um bloco de comando em que o predicado (*opt != END*) é testado, seu valor é verdadeiro, o

valor do atributo considerado é falso, não há chamadas empilhadas e o bloco possui a identificação *10*. Como qualquer alteração em um dos componentes causa a identificação de um contexto diferente, o contexto *4* possui dados similares ao *3*, mas o valor do predicado é verdadeiro neste ponto, indicando a execução do sistema dentro de um novo contexto. A Figura 4.3 apresenta um exemplo de uma tabela de Contextos.

Figura 4.3 – Exemplo de uma tabela de Contextos.

S	P	ID	V	A	
0	INITIAL	-1	T		<>
1	c.readCommand	0	T	{false}	<>
2	Integer.parseInt	1	T	{false}	<>
3	(opt != END)	10	T	{false}	<>
4	(opt != END)	10	F	{false}	<>
5	(RED)	9	1	{false}	<>
6	(isGreen)	8	F	{false}	<>
7	(GREEN)	9	0	{false}	<>
8	(! isGreen)	7	T	{false}	<>
9	call.TrafficLightsgreenLights	3	T	{false}	<>
10	TrafficLights.greenLights	11	T	{false}	<call.TrafficLightsgreenLights>
11	call.TrafficLightschangeColour	5	T	{true}	<call.TrafficLightsgreenLights, TrafficLights.greenLights>
12	TrafficLights.changeColour	13	T	{true}	<call.TrafficLightsgreenLights, TrafficLights.greenLights, call.TrafficLightschangeColour>
13	(opt != END)	10	T	{true}	<>
14	c.readCommand	0	T	{true}	<>
15	Integer.parseInt	1	T	{true}	<>

Fonte: (do autor , 2016).

Todos os rastros de um sistema sempre começam em um contexto *INITIAL* e que, em caso de contextos referentes a chamadas de métodos, o predicado associado é o nome do meu e seu valor é sempre definido como *true*. Desta forma, contextos permitem identificar em que situação determinados eventos ocorrem no sistema considerando-se a combinação das informações tabuladas. Os rastros de execução, detalhados a seguir, apresentam a ordem de ocorrência dos contextos de acordo com execuções do programa, incluindo ações locais ou entre componentes.

4.1.1.2 Arquivos de Registros de Execução

A execução do código instrumentado produz um arquivo de registro. A sequência de entradas é determinada por um caso de teste, o qual é capaz de forçar a execução de comportamentos que serão observados. Não é utilizada nenhuma técnica para selecionar um conjunto de testes, somente o conhecimento sobre o sistema e comportamentos que se deseja observar. Como resultado da execução do código-fonte instrumentado, geramos um conjunto de arquivos de registros, um para cada execução do sistema.

A Figura 4.4 apresenta um arquivo gerado após uma execução de um dos casos de estudo, o qual será detalhado a seguir, contendo uma sequência de ações e comandos de entrada, bem como chamadas de métodos.

Figura 4.4 – Exemplo de um arquivo de registro de execução.

```
CALL_ENTER:readCommand#TrafficLights=138093#c#{isGreen=false^}#0;
CALL_END:readCommand#TrafficLights=138093#c#0;
CALL_ENTER:parseInt#TrafficLights=138093#Integer#{isGreen=false^}#1;
CALL_END:parseInt#TrafficLights=138093#Integer#1;
SEL_ENTER:(GREEN)#0#TrafficLights=138093#{isGreen=false^}#9;
SEL_ENTER:(! isGreen)#true#TrafficLights=138093#{isGreen=false^}#7;
INT_CALL_ENTER:greenLights#TrafficLights=138093#{isGreen=false^}#3;
MET_ENTER:greenLights#TrafficLights=138093#{isGreen=false^}#11;
INT_CALL_ENTER:changeColour#TrafficLights=138093#{isGreen=true^}#5;
MET_ENTER:changeColour#TrafficLights=138093#{isGreen=true^}#13;
MET_END:changeColour#TrafficLights=138093#13;
INT_CALL_END:changeColour#TrafficLights=138093#5;
MET_END:greenLights#TrafficLights=138093#11;
INT_CALL_END:greenLights#TrafficLights=138093#3;
SEL_END:(! isGreen)#TrafficLights=138093#7;
SEL_END:(GREEN)#TrafficLights=138093#9;
REP_ENTER:(opt != END)#true#TrafficLights=138093#{isGreen=true^}#10;
CALL_ENTER:readCommand#TrafficLights=138093#c#{isGreen=true^}#0;
CALL_END:readCommand#TrafficLights=138093#c#0;
CALL_ENTER:parseInt#TrafficLights=138093#Integer#{isGreen=true^}#1;
CALL_END:parseInt#TrafficLights=138093#Integer#1;
REP_END:(opt != END)#TrafficLights=138093#10;
REP_ENTER:(opt != END)#false#TrafficLights=138093#{isGreen=true^}#10;
REP_END:(opt != END)#TrafficLights=138093#10;
```

Fonte: (do autor , 2016).

4.1.1.3 Rastos de Contextos

O processo de coletar informação dos rastros é feito com o objetivo de monitorar a execução do programa. Rastros de contextos de execução contém uma sequência de contextos do sistema e ações executadas entre eles. A Figura 4.5 apresenta um fragmento de um rastro de Contextos composto por contextos, representando o fluxo de controle do programa.

Os elementos denotados com o símbolo # são os contextos detalhados na tabela de contextos (Figura 4.3) e a sequência de contextos segue a ordem das informações armazenadas nos arquivos de registro de execução (Figura 4.4). Por exemplo, *#2* corresponde à entrada da tabela de contextos com o identificador igual a *2* e essa entrada segue a ordem escrita no arquivo de registro de execução. As entradas não precedidas pelo símbolo # representam ações, que podem ser chamadas de métodos internas, prefixadas com a palavra *call*; execução de métodos de outras classes, prefixadas com o identificador dessa outra classe (por exemplo *c*); ou ainda a execução do corpo de métodos (por exemplo *greenLights*), representadas através do nome do respectivo método. O sufixo *enter* denota a ação que

representa o início da execução da chamada ou do corpo de um método, enquanto o sufixo *exit* representa o respectivo término.

Figura 4.5 – Fragmento de um Rastro de Contexto.

```
#0
#1
c.readCommand.enter
c.readCommand.exit
#2
#7
#8
#9
call.greenLights.enter
#10
greenLights.enter
#11
call.changeColour.enter
#12
changeColour.enter
changeColour.exit
call.changeColour.exit
greenLights.exit
call.greenLights.exit
#13
#14
c.readCommand.enter
c.readCommand.exit
#15
```

Fonte: (do autor , 2016).

4.1.1.4 Geração da Gramática de Grafos

Considerando-se a abordagem de geração de modelos LTS a partir das informações dos contextos, propomos um mapeamento dos contextos para a criação de regras da gramática de grafos, no formato $r : L \rightarrow R$ definido anteriormente. Para esse mapeamento, elaboramos um algoritmo, apresentado na Figura 4.6, capaz de traduzir os rastros de contextos e a tabela de contextos para regras de gramática de grafos, conforme apresentado a seguir.

O algoritmo recebe como entrada uma tabela de contextos e os rastros de contextos, que representam execuções do programa. A partir dos rastros de contextos, identifica pares de contextos c_i e c_{i+1} que apareçam em sequência em algum rastro do conjunto *RC*, contendo ou não ações entre eles. Então, a análise é dividida em três casos: o primeiro, quando não há ocorrência de ações entre c_i e c_{i+1}; o segundo, quando existe uma única ação entre c_i e c_{i+1}; e o terceiro, quando há uma lista de ações entre c_i e c_{i+1}. Para o primeiro caso, somente criamos uma regra com o contexto c_i no lado esquerdo e o contexto c_{i+1} no lado direito. Na ocorrência do segundo caso, criamos a regra também com o contexto c_i no lado esquerdo e o contexto c_{i+1} no lado direito e adicionamos a ação existente entre os contextos c_i e c_{i+1} como mensagem no lado esquerdo ou no lado direito, de acordo com o tipo de informação que a ação representa. Finalmente, para o terceiro caso, onde ocorrem n ações entre c_i e c_{i+1}, são criadas $n+1$ regras, sendo que o contexto c_{i+1} somente aparece na última regra. As n primeiras

regras, nesse caso, possuem o contexto c_i no lado esquerdo e no lado direito, como se o contexto c_i fosse replicado, no rastro de execução, entre cada uma das ações.

Figura 4.6 – Algoritmo para construir uma gramática de grafos a partir de contextos.

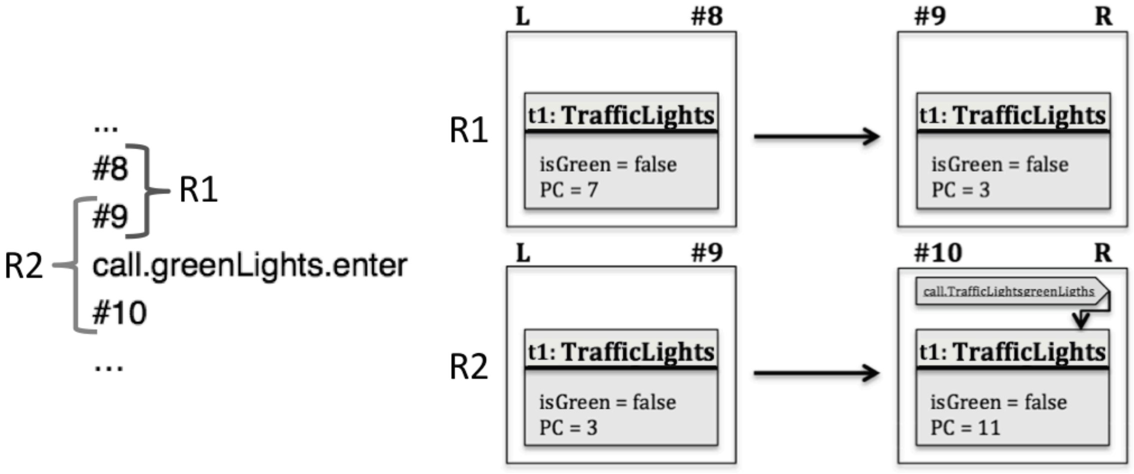
```
Entrada: TC: tabela de contextos, RC: conjuntos de rastros de contextos
Saída: GG: gramática de grafos
início
    GG = ∅;
    para cada par de contextos c_i, c_{i+1} ∈ RC faça
        se entre c_i e c_{i+1} não existe nenhuma ação então
            L ← c_i;
            R ← c_{i+1};
            ActionLeft ← ∅;
            ActionRight ← ∅;
            Cria uma nova regra r : L → R;
            GG ← GG + r;
        se entre c_i e c_{i+1} existe uma única ação act então
            L ← c_i;
            R ← c_{i+1};
            se act é mensagem enviada ou ação da mesma classe então
                se a pilha de métodos em execução de c_i ≠ ∅ então
                    ActionLeft ← primeiro elemento da pilha;
                senão
                    ActionLeft ← ∅;
                fim
                ActionRight ← act;
            se act é mensagem recebida ou ação de outra classe então
                ActionLeft ← act;
                ActionRight ← ∅;
            Cria uma nova regra r : L → R;
            GG ← GG + r;
        se entre c_i e c_{i+1} existem n ações então
            L_1 ← c_i;
            R_1 ← c_i;
            se a pilha de métodos em execução de c_i ≠ ∅ então
                ActionLeft_1 ← primeiro elemento da pilha;
            senão
                ActionLeft_1 ← ∅;
            fim
            ActionRight_1 ← act_1;
            Cria uma nova regra r_1 : L_1 → R_1;
            GG ← GG + r_1;
            para i = 2 até i < n faça
                L_i ← c_i;
                R_i ← c_i;
                ActionLeft_i ← act_{i−1};
                ActionRight_i ← act_i;
                Cria uma nova regra r_i : L_i → R_i;
                GG ← GG + r_i;
            fim
            L_n ← c_i;
            R_n ← c_i;
            ActionLeft_n ← act_{n−1};
            ActionRight_n ← act_n;
            Cria uma nova regra r_n : L_n → R_n;
            GG ← GG + r_n;
            L_{n+1} ← c_i;
            R_{n+1} ← c_{i+1};
            ActionLeft_{n+1} ← act_n;
            ActionRight_{n+1} ← ∅;
            Cria uma nova regra r_{n+1} : L_{n+1} → R_{n+1};
            GG ← GG + r_{n+1};
        fim
    fim
fim
```

Fonte: (do autor , 2016).

Ainda, é feita uma verificação na pilha de métodos em execução, que contém informações sobre métodos que foram invocados e ainda não terminaram sua execução, por exemplo, quando um método *A* faz uma chamada de um método *B* e aguarda o término da execução do método *B* para continuar com o sua própria execução. No segundo caso e na primeira regra do terceiro caso, quando essa pilha de métodos não está vazia o primeiro item da pilha é colocado como mensagem no lado esquerdo da regra.

A Figura 4.7 apresenta duas regras que ilustram o funcionamento do algoritmo descrito acima, utilizando a tabela de contextos e os rastros de contextos apresentados anteriormente. No primeiro caso, a mudança do contexto *#8* para o *#9*, sem nenhuma mensagem entre eles, resulta em uma regra simples, onde o lado esquerdo representa o primeiro contexto do par, e o lado direito o segundo. Enquanto que para o segundo par de contextos, temos uma única ação, mapeada para a regra na forma de mensagem, de acordo com o algoritmo.

Figura 4.7 – Exemplo do funcionamento do algoritmo para mapeamento dos contextos para regras da gramática de grafos, nos dois primeiros casos do algoritmo.



Fonte: (do autor , 2016).

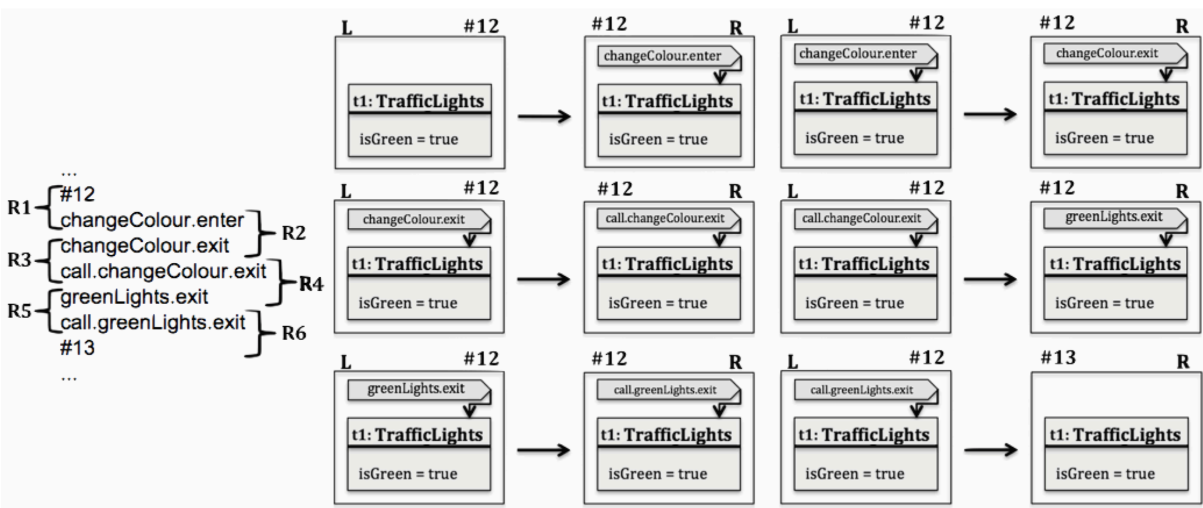
Quando existem múltiplas ações entre um par de contextos o algoritmo realiza a geração das regras de outra forma, conforme detalhado anteriormente, onde as *n* ações existentes entre um par de contextos são representadas por um conjunto de *n+1* regras. Essa situação é retratada na Figura 4.8.

O algoritmo mapeia cada rastro de contextos para uma sequência de regras que descrevem exatamente aquele rastro, com um certo nível de abstração. Os rastros são extraídos da execução do programa, logo, eles representam sempre comportamentos válidos

desse sistema. Assim, a combinação de múltiplos rastros retém os comportamentos isolados originais e não inclui qualquer comportamento inválido no nível de abstração utilizado.

Considerando que o algoritmo possui uma complexidade linear, sua performance depende da cardinalidade do conjunto de rastros de contextos recebido como entrada. O algoritmo trabalha com pares de contextos consecutivos nos rastros, portanto, quanto maior o número de rastros, maior o número de regras geradas. Posteriormente, são discutidas algumas estratégias para otimização da geração de regras.

Figura 4.8 – Exemplo do funcionamento do algoritmo para mapeamento dos contextos para regras da gramática de grafos quando existem múltiplas ações entre um par de contextos.



Fonte: (do autor , 2016).

4.1.2 Casos de Estudo

Para avaliarmos a aplicabilidade e os resultados da abordagem proposta, a mesma foi aplicada a duas aplicações simples, denominados casos de estudo, detalhados a seguir.

4.1.2.1 Traffic Lights

Nosso primeiro caso de estudo chama-se *Traffic Lights*, cujo código-fonte é apresentado na Figura 4.9. Essa aplicação simula o funcionamento de um semáforo com uma luz verde e uma vermelha, alternando o acendimento das luzes de cores distintas. A classe *TrafficLights* possui somente uma variável booleana, chamada *isGreen*, utilizada para identificar quando a luz verde está ligada. Quando a variável que representa a luz verde está inativa considera-se que a luz vermelha está ligada. Esse sistema possui somente uma classe com quarenta linhas de código-fonte.

Figura 4.9 – Código-fonte Java da classe *TrafficLights*.

```
import java.io.IOException;
class TrafficLights {
      private static final int GREEN = 0;
      private static final int RED = 1;
      private static final int END = 2;
      private static boolean isGreen;

  public TrafficLights (CommandReader c) {
    isGreen = false;
    int opt = - 1;
    do {
      try {
          opt = Integer.parseInt(c.readCommand ());
      } catch (IOException e) {
          System.out.println(e.getStackTrace());
      }
      switch (opt) {
      case GREEN :
          if (!isGreen)
               greenLights ();
         break;

        case RED :
          if (isGreen)
               redLights ();
         break;
      }
    } while (opt != END);
  }

  private void greenLights () {
      isGreen = true;
      changeColour ("green");
  }

  private void redLights () {
    isGreen = false;
    changeColour ("red");
  }

  private void changeColour (String newColour) {
      System.out.println (newColour);
  }
}
```

Fonte: (do autor , 2016).

4.1.2.2 Air Conditioner

A segunda aplicação utilizada como caso de estudo chama-se *Air Conditioner*. Essa aplicação simula a operação de um condicionador de ar, responsável por controlar a temperatura de um ambiente através de sensores. Esses sensores são representados por quatro variáveis booleanas (*roomHot*, *doorClosed*, *acOn*, *SystemOff*) que representam a situação dos elementos do ambiente, como temperatura, portas e funcionamento do sistema. Essa aplicação

é composta por quatro classes, totalizando 120 linhas de código-fonte, apresentado na Figura 4.10.

Figura 4.10 – Código-fonte Java da classe *AirConditioner*, uma das quatro classes da aplicação.

```
class AirConditioner implements Signals {

  private static boolean room_hot;
  private static boolean door_closed;
  private static boolean ac_on;

  public AirConditioner (EnvController c) {
    room_hot = false;
    door_closed = true;
    ac_on = false;

    boolean finished = false;
    int message = - 1;

    while (! finished) {
      message = c.nextSignal ();

      switch (message) {
        case ROOM_HOT:
          if (!room_hot) {
            room_hot = true;        #action:"roomHot";
             System.out.println ("-> Room hot");
          }
          break;

        case ROOM_COOL:
          if (room_hot) {
            room_hot = false;       #action:"roomCool";
             System.out.println ("-> Room cool");
          }
          break;

        case DOOR_OPEN:
          if (door_closed) {
            door_closed = false;          #action:"doorOpen";
            System.out.println ("-> Door open");
            if (ac_on) {
              ac_on = false;        #action:"acOff";
              System.out.println ("-> AC off");
            }
          }
          break;

        case DOOR_CLOSED:
          if (!door_closed) {
            door_closed = true;           #action:"doorClosed";
            System.out.println ("->Door closed");

            if (room_hot) {
              ac_on = true;               #action:"acOn";
              System.out.println ("-> AC on");
            }
          }
          break;

        case OFF:
          finished = true;                #action:"finished";
          break;

        default:
          System.out.println ("Incorrect command!");
      }
    }
  }
}
```

Fonte: (do autor , 2016).

4.1.3 Experimentos Preliminares

Para os dois casos de estudo, o algoritmo foi aplicado e foi gerada uma gramática de grafos para cada sistema. A Tabela 4.1 apresenta os resultados do processo de extração para cada aplicação. Para o primeiro caso de estudo, *Traffic Lights*, obtivemos um total de 38 regras, enquanto que para a segunda aplicação, *Air Conditioner*, 103 regras foram geradas.

Tabela 4.1 - Resultados da extração de Gramática de Grafos para os casos de estudo.

Aplicação	Regras Geradas	Código-fonte	Classes
Traffic Ligths	38	40 linhas	1
Air Conditioner	103	120 linhas	4

A partir desses dados podemos notar um elevado número de regras geradas, considerando-se que as aplicações utilizadas como casos de estudo são simples e pequenas. Foi identificado que esse número excessivo de regras é devido ao fato de que a maioria das mudanças de contextos estão relacionadas unicamente ao fluxo de controle do sistema, ou seja, não efetuam alterações nos valores dos atributos do sistema, nos dados da aplicação. Podemos visualizar essa situação, por exemplo, na Figura 4.7, onde a regra *r1* representa somente a mudança no fluxo de controle do programa, sem nenhuma mensagem entre os contextos que aparecem na regra nem mudança nos valores dos atributos da classe, unicamente para incrementar o valor do contador do programa, ou *Program Counter* (*PC*).

Analisando os resultados preliminares, foi identificada a necessidade de melhorias na abordagem, afim de possibilitar a geração de gramáticas de grafos com um número menor de regras, porém mantendo sua representatividade. Além do número excessivo de regras possivelmente prejudicar futuras análises, pois dificulta a manipulação da gramática, foi observado que as regras possuem um certo excesso de informações, como o fluxo de controle. Em vista dessa situação, desenvolvemos estratégias para refinar e melhorar o processo de extração, apresentadas a seguir.

4.1.4 Otimizações na Geração da Gramática de Grafos

A partir das dificuldades relatadas previamente, elaboramos algumas estratégias com o objetivo de resolver, ou ao menos minimizar, o problema da geração de um grande número de regras durante o processo de extração de uma gramática de grafos. Uma vez que o formalismo de Gramática de Grafos é dirigido a dados, as informações de fluxo de controle presentes nas regras introduzem uma sequencialidade indesejada no conjunto de regras. Essa sequencialidade não é natural em GG, pois a ausência de sequências forçadas e a

possibilidade de execução paralela de regras não-conflitantes é característica desse formalismo. Portanto, eliminando essa informação é possível reduzir o número das regras, mantendo no modelo final somente as regras que contém informações relacionadas aos dados, sem alterar a semântica do modelo. Além do *PC*, foram analisadas também as mensagens presentes nas regras, com o objetivo de eliminar as regras que possuem mensagem em somente um dos lados, e as marcações de chamadas de métodos, através dos sufixos *enter* e *exit*, a fim de sintetizar chamadas de métodos e a execução do corpo do método em uma única regra.

4.1.4.1 Program Counter

Analisando o conjunto de regras gerados, percebemos que, em algumas regras, a única diferença entre o lado esquerdo e o lado direito da regra era o valor do contador de programa abstrato, o *Program Counter*. Uma vez que utilizamos um formalismo dirigido a dados, excluímos as regras que não realizam alterações nos dados do programa, como a regra *r1* na Figura 4.7. Considerando isso, nós alteramos a estrutura das regras geradas, eliminando a informação sobre o fluxo de controle do programa, expressa através do PC. Assim, no momento da criação das regras, o valor do PC é ignorado e, quando são identificadas regras idênticas desconsiderando o PC, desprezamos essas regras, evitando, assim, duplicações.

4.1.4.2 Análise de Mensagens

Mensagens são utilizadas na comunicação entre objetos. Na abordagem proposta, as mensagens são representadas por nós específicos nos grafos da gramática. Analisando as informações das mensagens podemos sintetizar a gramática gerada, concatenando regras que simbolizam somente o envio ou o recebimento de uma mensagem, sem afetar os dados do sistema. Por exemplo, sempre que duas regras possuem os mesmos valores de atributos, sem nenhuma mudança nesses dados e não possuem mensagens no lado direito da primeira regra nem no lado esquerdo da segunda regra, podemos concatenar essas regras em uma nova regra, com o lado esquerdo da primeira regra e o lado direito da segunda. Observando as regras da Figura 4.7, se não tivermos nenhuma mensagem no lado direito da regra *r1*, podemos criar uma regra com o lado esquerdo da regra *r1* e com o lado direito da regra *r2*. Assim, diminuímos o número de regras através de sucessivas concatenações.

4.1.4.3 Chamadas de Métodos

Os rastros utilizados para gerar uma GG contém marcações nos nomes das ações que representam chamadas de métodos, na forma de sufixos *enter* e *exit*. Essas marcações estão presentes porque auxiliam a identificação do início e final da chamada ou execução de um método, respectivamente.

Utilizando a ferramenta AGG (TAENTZER, 2004), podemos criar regras concorrentes que representam a aplicação de sequências de regras. A regra concorrente descreve um estado parcial do sistema antes da aplicação de uma sequência de regras e o estado resultante da aplicação dessas regras. Assim, a criação de regras concorrentes pode reduzir substancialmente o número de regras no conjunto final.

Para cada chamada de método, composta por duas mensagens com os sufixos *enter* e *exit*, criamos uma sequência com as regras que são executadas entre a mensagem inicial e a final e geramos uma regra concorrente que representa essa sequência, removendo as regras originais do conjunto. A aplicação da regra concorrente tem o mesmo efeito da aplicação de todas as regras pertencentes a sequência que originou a regra concorrente. Sendo assim, o lado esquerdo da regra concorrente possui uma ação com o sufixo *enter* e o lado direito da regra possui uma ação com o sufixo *exit*.

4.2 Resultados

Após a definição das estratégias de otimização, o algoritmo foi aplicado novamente aos casos de estudo e foram obtidos novos resultados, apresentados na Tabela 4.2, onde é relatado o número de regras geradas após a aplicação de cada um dos métodos de redução da gramática. É importante notar que, apesar das nossas estratégias terem como objetivo principal a redução do número de regras, em alguns casos é necessário criar novas regras no nosso conjunto final, tendo essas regras o papel de representar aquelas que foram eliminadas. Outro aspecto importante a se observar é que excluindo regras da gramática de grafos, possivelmente uma ou mais regras que possuem *match* com a regra excluída podem precisar de ajustes.

Tabela 4.2 - Resultado da extração de Gramática de Grafos após a aplicação das estratégias de otimização do algoritmo.

Aplicação	Regras Geradas	PC	Análise das Mensagens	Chamada de Métodos
Traffic Ligths	38	25	21	9
Air Conditioner	103	74	56	36

Aplicando o primeiro refinamento, relacionado à análise do efeitos das regras sobre os dados, o número de regras para os casos de estudo foi reduzido para um total de 28 regras no primeiro caso de estudo e 77 regras no segundo. Dessa forma, alcançamos uma redução de até 25% do número inicial de regras no conjunto final. Após a análise das mensagens, obtivemos uma maior redução no número de regras, diminuindo para 21 regras para o primeiro caso de estudo e para 61 regras para o segundo. Finalmente, aplicando a terceira estratégia de otimização do conjunto de regras, para a aplicação *Traffic Lights* obtivemos um número final de 13 regras, enquanto que o conjunto de regras da aplicação *Air Conditioner* se reduziu a 50 regras.

Analisando os resultados obtidos, é possível observar que a redução obtida no primeiro caso de estudo, aproximadamente 65%, foi relativamente maior do que no segundo caso, aproximadamente 50%. Essa diferença existe em razão do grande número de chamadas de métodos aninhadas, uma vez que na primeira aplicação essa situação aparece com maior frequência. Uma vez que utilizamos um modelo que contem informações sobre o fluxo de controle do sistema como base para geração da gramática e as chamadas de métodos aparecem nos rastros de execução, quando mais chamadas de métodos aparecerem nos rastros, maior será o número de regras geradas.

Comparando a gramática gerada inicialmente com a gramática obtida após os refinamentos do conjunto de regras, é possível observar uma significativa redução nos dois casos. É importante notar, ainda, que essa redução no número de regras não afeta o comportamento descrito pelo modelo, pois as regras refinadas são somente mais abstratas que as regras originais.

4.3 Trabalhos Relacionados

Na literatura, existem alguns trabalhos que exploram a extração de modelos de diversas formas. Com relação à extração de modelos comportamentais, existem algumas abordagens que utilizam informações estáticas (HOLZMANN, SMITH, 1999) (CORBETT ET AL. 2000) (HENZINGER ET AL. 2002) (BALL, RAJAMANI 2002) (CHAKI ET AL., 2005), outras que utilizam informações dinâmicas (COOK, WOLF, 1998) (LORENZOLI, MARIANI, PEZZE, 2006) (COMPARETTI ET AL., 2009) e, ainda, que utilizam informações híbridas (NIMMER, ERNST, 2002). Todas essas abordagens utilizam modelos baseados em fluxo de controle ou em alguma sequência de eventos. A abordagem descrita em (DUARTE, KRAMER, UCHITEL, 2006) propõe a combinação desses dois elementos além da utilização de informações sobre valores de atributos do sistema. Dessa forma, essa

abordagem difere das anteriores porque tem como objetivo a obtenção de um modelo dirigido a dados, utilizando uma abordagem híbrida.

Considerando a pesquisa especificamente acerca da extração de Gramática de Grafos, em (ZHAO, KONG, ZHANG, 2010) é descrita uma abordagem que utiliza inferência de gramática em rastros de execução para a criação de grafos de chamadas de métodos aninhadas hierarquicamente, a partir de *bytecode* Java, com o objetivo de verificar propriedades no grafo gerado, o qual representa sequências válidas de chamadas. Nesse caso, sequências inválidas representam sequências de chamadas potencialmente perigosas para o comportamento do sistema. Embora a abordagem proposta neste capítulo também utilize rastros de execução para a geração da Gramática de Grafos, as informações não se restringem a sequências de chamadas de métodos, uma vez que são utilizadas também as informações sobre os valores dos atributos do sistema a cada ponto de execução. A partir dessa informação, são definidos estados abstratos do sistema, baseados também em dados, o que proporciona a exploração de características do formalismo de grafos, uma vez que esse modelo resultante contém mais informações, além do fluxo de controle.

Uma das abordagens mais próximas da proposta nesse capítulo é apresentada em (CORRADINI, 2004). Os autores propõem uma abordagem para a construção de Gramática de Grafos a partir de programas na linguagem de programação Java. No entanto, a metodologia utilizada para a tradução de Java para o formalismo de grafos é baseada somente no código-fonte e na gramática da linguagem Java. Além disso, essa tradução pode ser aplicada a um conjunto limitado de instruções da linguagem, ou ainda, requer algumas conversões entre instruções, como, por exemplo, o comando de repetição *while*, que precisa ser substituído por chamadas a funções recursivas.

Na abordagem proposta por este trabalho, não é necessário nenhuma modificação no código-fonte original, somente são introduzidas algumas anotações necessárias para a coleta de informações. Além disso, a combinação das informações obtidas a partir do código-fonte, como o fluxo de controle, com as informações retiradas a partir da observação do comportamento do software, através dos rastros de execução, proporciona uma melhor compreensão sobre o comportamento do sistema e resulta na criação somente de regras que realmente podem ocorrer. Por outro lado, esse modelo contém somente regras que foram inferidas a partir dos rastros de execução observados, portanto não é possível garantir que os comportamentos observados representam todos os comportamentos possíveis do software.

Outro trabalho que se aproxima ao desenvolvido nesta pesquisa é apresentado em (ALSHANQITI, HECKEL, 2014), onde os autores apresentam uma técnica para extração de

regras a partir da observação das transformações entre o estado inicial e o estado final do sistema, baseada somente nos rastos de execução. O objetivo nesse trabalho não é gerar um modelo a partir de código-fonte Java, mas sim gerar um modelo para explicar um conjunto de observações. Assim, busca-se a criação de um modelo que possa melhorar a compreensão do sistema, considerando pré e pós-condições para a execução dos métodos. Todas as pré-condições tornam-se o lado esquerdo da regra e as pós-condições o lado direito. Para restringir o conjunto de objetos incluídos em cada regra, é realizada a identificação de quais objetos do sistema são afetados e/ou requeridos para que a regra em questão seja aplicada. A abordagem proposta neste capítulo diferencia-se do trabalho descrito em (ALSHANQITI, HECKEL, 2014) na medida em que neste trabalho é seguida a ideia de encapsulamento, pois, somente o estado do sistema para o qual a regra está sendo criada é visível e somente são incluídos na regra em questão os objetos referenciados pelo objeto principal, necessários para a descrição de um estado do sistema ou uma mudança no estado. Dessa forma, pretende-se que, seja possível descrever o comportamento do sistema e aplicar análises sobre as gramáticas, no intuito de detectar potenciais problemas e auxiliar a compreensão do software.

5 EXTRAÇÃO DE GRAMÁTICA DE GRAFOS A PARTIR DE CASO DE USO

Neste capítulo, é apresentada uma metodologia para a extração e verificação de modelos especificados através do formalismo de gramática de grafos a partir de Casos de Uso, ou seja, partindo da documentação do software, previamente à implementação. É realizada uma validação da metodologia apresentada, através de um experimento com artefatos de um projeto de software real e, por fim, é explorada uma possível extensão da metodologia, onde são propostas análises complementares sobre sistemas modelados através de gramática de grafos.

Casos de Uso, ou *Use Cases (UC)*, são um modelo bastante comum e muito utilizados para documentação de comportamentos esperados de um software. Eles podem ser usados em diferentes processos de software, não somente na documentação e validação de requisitos, mas também na especificação de *design*, verificação e evolução de um sistema. Esse tipo de documentação é um ponto de referência importante no processo de desenvolvimento de software.

Na prática, a descrição de um caso de uso é tipicamente informal, fazendo uso, na maiorias das vezes, de linguagem natural dentro de uma estrutura predefinida. Uma vez que essa descrição é informal, a mesma pode conter problemas inerentes à linguagem, como ambiguidade e imprecisão, por exemplo. Essa situação pode, ainda, levar a um aumento no número de problemas quando pequenas incertezas são propagadas para outras fases do processo de desenvolvimento, prejudicando a qualidade do sistema como um todo.

De fato, é comum que a maioria das falhas no software sejam introduzidas durante a fase de especificação. No entanto, é importante que as descrições dos casos de uso sejam mantidas em um formato familiar às pessoas envolvidas no projeto, chamados *stakeholders*, uma vez que os mesmos atuam diretamente na definição dos casos de uso. Sendo assim, a verificação de casos de uso, normalmente, corresponde a inspeções manuais no software, como *walkthroughs* (MYERS, SANDLER, BADGETT, 2011), onde a detecção de problemas como incompletude e ambiguidade não é uma tarefa trivial.

Em vista dessas circunstâncias, foi desenvolvida uma estratégia para formalização dos casos de uso, apresentada em (RIBEIRO ET AL., 2014), a qual utiliza o formalismo de transformação de grafos para modelar e analisar casos de uso. Existem diferentes estratégias para formalização de casos de uso, no entanto, em sua maioria, as mesmas determinam uma sintaxe específica para a descrição de um caso de uso, o que muitas vezes limita a expressividade dos requisitos em relação a linguagem utilizada pelos *stakeholders*, e podem

restringir a semântica do caso de uso. No entanto, a metodologia a ser explorada neste trabalho, diferencia-se das demais devido a utilização de gramática de grafos e das características desse formalismo, uma linguagem visual com uma semântica simples e, ao mesmo tempo, muito expressiva, possibilitando a realização de análises sobre os modelos gerados.

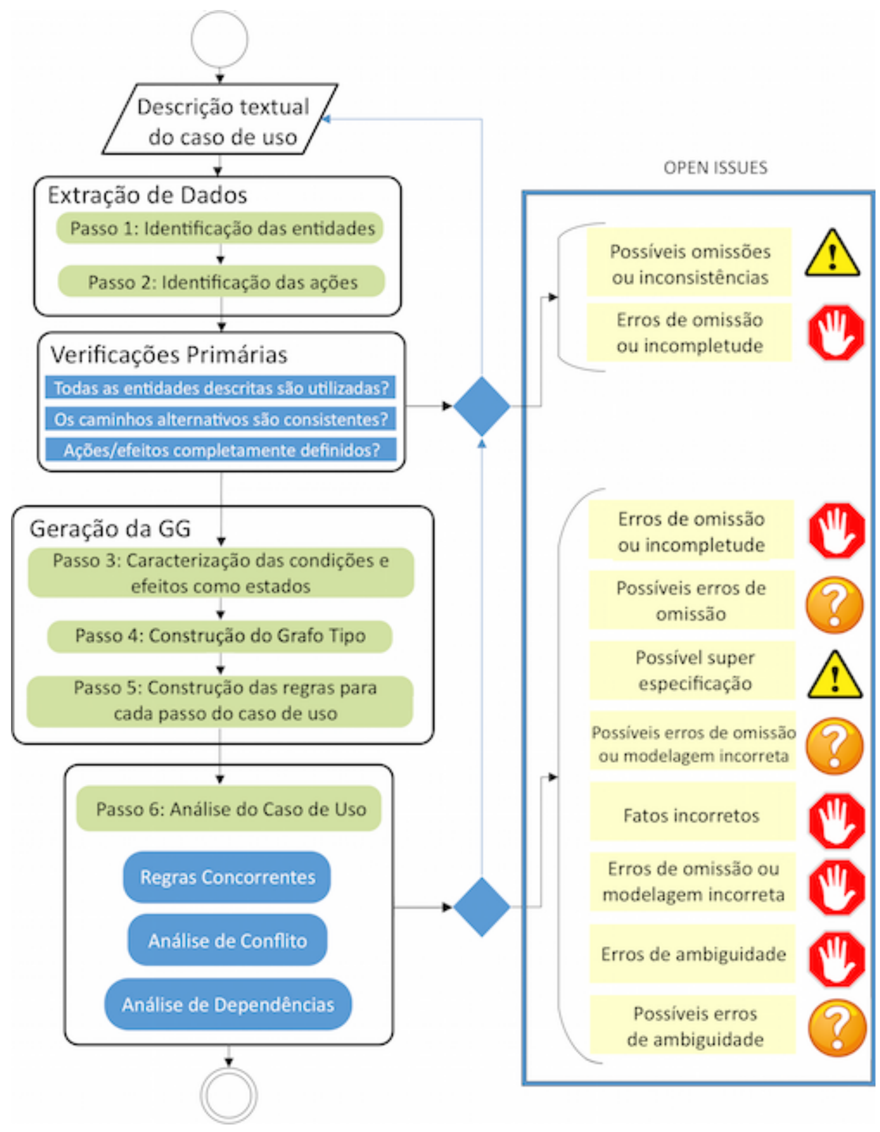
Essa metodologia tem como objetivo integrar a formalização dos casos de uso com análises sobre os modelos, suportadas por ferramentas, possibilitando assim um aumento na qualidade dos casos de uso. O processo de formalização é composto por uma sequência de passos para guiar a construção de um modelo formal, executar análises e avaliar os resultados em termos dos níveis dos problemas encontrados. A seguir, a metodologia é apresentada, de forma sucinta, juntamente com um exemplo de sua utilização.

5.1 Metodologia para Formalização de Casos de Uso

A estratégia proposta em (RIBEIRO ET AL., 2014), para a formalização de casos de uso, é dividida em quatro fases principais, como mostra a Figura 5.1. Partindo de uma descrição textual de um caso de uso, a primeira fase (*Extração de dados*) consiste em identificar entidades (*Passo 1*) e ações (*Passo 2*) que farão parte do modelo formal, juntamente com um conjunto de *Verificações Primárias*, relacionadas com a consistência das informações extraídas. São observadas inconsistências que podem afetar o modelo ou até mesmo identificadas entidades ou ações que são mencionadas na descrição textual mas não são utilizadas, ou seja, não estão, portanto, claramente definidas. Nos casos onde são encontradas inconsistências, o caso de uso pode ser reescrito para eliminá-las ou o analista pode anotá-las como possíveis problemas, para serem resolvidos posteriormente.

Após as verificações primárias, inicia-se a fase de geração de uma gramática de grafos (*Geração da GG*). Nesse processo, condições e efeitos das ações são modelados como estados (grafos) no *Passo 3*. No *Passo 4*, um grafo tipo é construído através da definição da representação gráfica dos artefatos gerados no primeiro e terceiro passos. Na penúltima etapa, o *Passo 5*, cada passo da descrição do caso de uso é modelado como uma regra de transição entre estados (grafos) do sistema, usando as estruturas definidas nos passos 3 e 4. Finalmente, com o conjunto de regras e o grafo tipo criado, ou seja, uma gramática de grafos, é possível realizar uma série de verificações automáticas, baseadas em *regras concorrentes* e nas análises de *conflitos* e *dependências*.

Figura 5.1 – Visão geral da estratégia para formalização e verificação de casos de uso através do formalismo de transformação de grafos.



Fonte: Ribeiro et al. (2012).

Realizando as análises com auxílio da ferramenta AGG (TAENTZER, 2004) podemos detectar possíveis problemas no modelo gerado, os quais são anotados sob a denominação de *Open Issues (OI)* e classificados de acordo com o nível de gravidade de cada um, podendo ser: amarelo (⚠), significando um problema simples, que possivelmente poderá ser resolvido pelo próprio analista; laranja (?) que representa um problema de gravidade intermediária, o qual provavelmente exigirá uma confirmação dos *stakeholders*; e vermelho (✋), que sinaliza um problema mais grave, e poderá implicar em uma modificação na descrição do caso de uso. Qualquer decisão de design feita a partir de uma OI deve ser documentada, pois uma vez que as soluções podem implicar em alterações no caso de uso é importante que as informações

originais do caso de uso sejam preservadas para, quando necessário, retornar à situação inicial.

Através das análises, é possível verificar, por exemplo, se as pré e pós-condições estão corretamente definidas e se existem conflitos e/ou dependências entre regras da gramática, ou seja, entre passos do caso de uso. Os passos do processo de formalização atuam como um roteiro para o analista e são detalhados a seguir. O Caso de Uso *Login*, apresentado na Figura 5.2, simula a operação de *login* de um usuário em um caixa eletrônico de um sistema bancário, e será utilizado como exemplo para a demonstração do funcionamento da metodologia.

Figura 5.2 – Caso de uso *Login*, utilizado como exemplo.

Especificação do Caso de Uso (original)

Número	1
Nome	Login no Caixa Eletrônico
Resumo	Usuário realiza login no caixa eletrônico
Prioridade	5
Pré-condições	Usuário possui um cartão bancário e uma senha registrados
Pós-condições	Usuário visualiza o menu de operações do Caixa Eletrônico
Ator(es) Primário(s)	Cliente do Banco
Ator(es) Secundário(s)	Banco de Dados com as contas dos clientes

Gatilho	Única opção no caixa eletrônico	
Caminho Principal	**Passo**	**Ação**
	1	Sistema pede um cartão do banco
	2	Usuário insere um cartão
	3	Sistema pede uma senha
	4	Usuário insere a senha
	5	Sistema valida o cartão e a senha do usuário e mostra o menu de operações
Extensões	**Passo**	**Caminhos Alternativos**
	5a	Sistema notifica o usuário que o cartão é inválido
	5b	Sistema vai para a opção de saída
Anotações		

Fonte: (do autor , 2016).

A seguir, são descritas brevemente as etapas da metodologia. Detalhes sobre a criação da gramática, estudos preliminares, bem como instruções para realização e a exploração das análises propostas, de forma aprofundada, encontram-se no Anexo I.

1. **Identificação das entidades**: Primeiramente, o analista deve identificar na descrição do caso de uso todas as entidades envolvidas. Por exemplo, para o caso de uso utilizado como exemplo temos as seguintes entidades: *Usuário, Caixa Eletrônico, Sistema, Cartão Bancário* e *Senha*.
2. **Identificação das ações**: Esse passo define uma *Tabela de Ações*, contendo uma entrada para cada ação do caso de uso, onde são detalhadas as entidades envolvidas, as condições e os efeitos de cada ação. Nesse passo, também é criada uma *Tabela de Caminhos Alternativos*, onde são descritos os ponto de decisão do sistema e apresentados os caminhos que poderão ser seguidos, juntamente com as condições do sistema que validam esse caminho. Nessa etapa são realizadas algumas verificações para identificar se os efeitos estão claramente definidos, se existem entidades não utilizadas e se os caminhos alternativos são utilizados.
3. **Modelagem das condições e efeitos como estados**: Nesse passo é definido explicitamente como as condições e efeitos listados na *Tabela de Ações* e as pré e pós-condições do caso de uso devem ser representados através de nós e arestas de um grafo, construindo uma tabela, chamada *Tabela de Condições/Efeitos*. Ainda, construímos uma *Tabela de Operações*, com duas operações predefinidas, *Input* e *Output*, que representam as operações de entrada e saída do sistema, respectivamente.
4. **Construção do Grafo Tipo**: Utilizando as entidades, listadas no primeiro passo, e as tabelas de condições/efeitos e operações construídas no terceiro passo, constrói-se nessa etapa um grafo tipo. O grafo tipo é composto por todos os nós e arestas necessário para caracterizar os efeitos e as condições dos passos do caso de uso. Para um determinado nó ou aresta ser utilizado em uma regra, obrigatoriamente, esse elemento deverá estar representado no grafo tipo.
5. **Construção das Regras:** As regras que formalmente descrevem o comportamento do caso de uso são construídas nesse passo. Para cada ação, listada na *Tabela de Ações*, é construída uma regra contendo, em seu lado esquerdo, um grafo que descreve as condições que devem existir para aquela ação ocorrer. Esses grafos correspondem a cada condição previamente descrita na *Tabela de Condições/Efeitos*,

uma vez que somente é necessário uni-las apropriadamente. Analogamente, o lado direito de cada regra é construído usando os efeitos de cada ação.

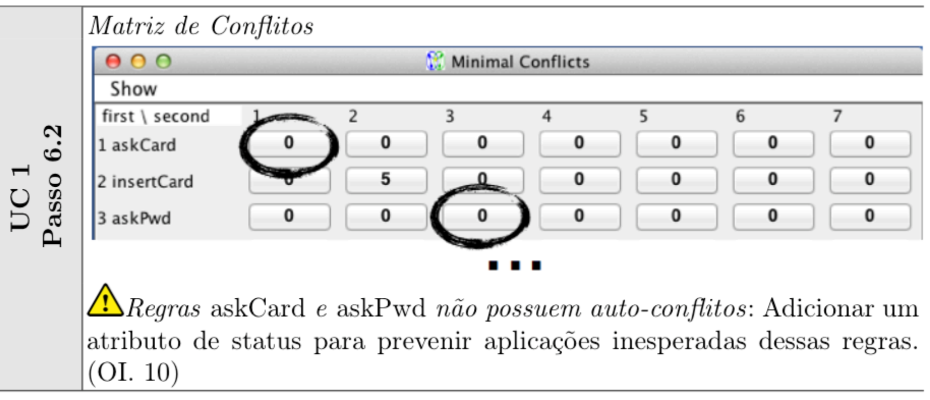
6. **Análise do Caso de Uso:** Nesta etapa, são realizadas algumas análises sobre o modelo formal extraído a partir da descrição de um caso de uso. As análises são realizadas com auxílio de ferramentas e podem ser realizadas em qualquer ordem, uma vez que os seus resultados são complementares.

   6.1. **Efeito do Caso de Uso:** Para fazer a análise de efeito do caso de uso, primeiramente, é necessário definir sequências de regras (*Rule Sequences (RSs)*) que representam a execução de todos os caminhos possíveis do caso de uso. Uma sequência de regras é definida por um conjunto de regras, construídas no passo 5, que implementa a execução de um cenário do caso de uso. Baseado em cada sequência de regra possível, construímos uma única regra, chamada *Regra Concorrente*. A regra concorrente simula a aplicação de toda uma sequência de regras, portanto, uma vez que a sequência de regras gerada simula um cenário possível do caso de uso, a respectiva regra concorrente mostra o estado do sistema antes e depois da aplicação dessa sequência, possibilitando a análise do real efeito do caso de uso no sistema. Essa análise deixa explícito as condições e os efeitos do caso de uso. Tudo que é necessário para a execução do caso de uso (pré-condições) deve estar presente no lado esquerdo da regra concorrente e, da mesma forma, o efeito do caso de uso no sistema (pós-condições) deve estar retratado no lado direito da regra. A construção da regra concorrente é feita através das dependências entre as regras de uma sequência, portanto, quando não é possível construir uma regra concorrente possivelmente há alguma omissão de informações em uma ou mais regras, como, por exemplo, a multiplicidade dos elementos do grafo.

   6.2. **Análise de Conflitos (Pares Críticos):** A análise de pares críticos sobre os conflitos mostra ao analista quais as regras são mutuamente exclusivas, ou seja, os pontos de decisão do sistema. O resultado da análise de conflitos é uma *Matriz de Conflitos*, onde as linhas e colunas são representadas pelas regras e cada célula é preenchida com um número, que indica quantos itens de uma regra estão em conflito com itens da outra regra. Os conflitos acontecem somente no lado esquerdo das regras e quando não existem conflitos o valor da célula correspondente é zero. Através da análise dos conflitos, é possível observar se regras que correspondem a um ponto de decisão realmente estão em conflito, no

modelo gerado, e, vice-versa, se conflitos que aparecem no modelo estão retratados como pontos de decisão do sistema. A solução para situações como essas depende exclusivamente do artefato que se está modelando, portanto, não é possível generalizar uma definição de quando um conflito é certo ou errado. Cada caso necessita de uma análise, com o intuito de observar se o comportamento do modelo gerado está coerente com o comportamento desejado do sistema. Um exemplo de matriz de conflito é apresentado na Figura 5.3.

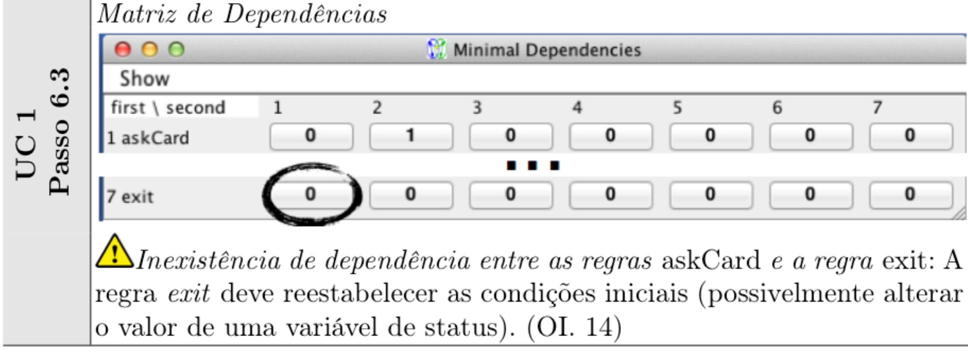
Figura 5.3 – Resultado da aplicação do Passo 6.2 ao caso de uso *Login*.



Fonte: (do autor , 2016).

6.3. **Análise de Dependências (Pares Críticos):** Análogo a análise de conflitos, a análise de pares críticos sobre dependências é feita através da construção de uma matriz, a *Matriz de Dependências*, da qual é apresentado um exemplo na Figura 5.4. Essa análise relaciona o lado direito de uma regra com o lado esquerdo de outra regra e, nas células da matriz, estão indicados quantos itens da segunda regra são dependentes da primeira. Da mesma forma, quando não há dependência entre duas regras o valor apresentado na célula é zero.

Figura 5.4 – Resultado da aplicação do Passo 6.3 ao caso de uso *Login*.



Fonte: (do autor , 2016).

Essa matriz mostra o relacionamento entre regras e pode ser utilizada para verificar se as dependências intuitivamente esperadas realmente ocorrem. Se duas regras, as quais devem ser executadas em uma determinada ordem, são mostradas como independentes na matriz, ou seja, podem acontecer em qualquer ordem, pode ser desejável inserir alguma forma de gerar uma dependência entre as mesmas.

Como resultado final desse processo obtemos uma descrição textual do caso de uso mais precisa e completa, como mostra a Figura 5.5. Os passo iniciais do processo podem ser feitos manualmente, sem a utilização de nenhuma ferramenta pelo analista. Entretanto, nos passos 3, 4 e 5, o analista deve fazer uso de ferramentas como a AGG, a qual auxilia na construção de um modelo formal do sistema, além de proporcionar uma representação gráfica dos grafos. Essa ferramenta é também muito importante para o passo 6, as análises sobre o modelo, pois proporciona a geração automática de regras concorrentes e a computação das matrizes de conflitos e dependências, processos esses que não são fáceis de se executar manualmente.

Figura 5.5 – Descrições do caso de uso antes (esquerda) e depois (direita) da aplicação da metodologia para formalização e verificação de casos de uso.

Especificação do Caso de Uso (original)

Número	1
Nome	Login no Caixa Eletrônico
Resumo	Usuário realiza login no caixa eletrônico
Prioridade	5
Pré-condições	Usuário possui um cartão bancário e uma senha registrados
Pós-condições	Usuário visualiza o menu de operações do Caixa Eletrônico
Ator(es) Primário(s)	Cliente do Banco
Ator(es) Secundário(s)	Banco de Dados com as contas dos clientes

Gatilho	Única opção no caixa eletrônico	
Caminho Principal	**Passo**	**Ação**
	1	Sistema pede um cartão do banco
	2	Usuário insere um cartão
	3	Sistema pede uma senha
	4	Usuário insere a senha
	5	Sistema valida o cartão e a senha do usuário e mostra o menu de operações
Extensões	**Passo**	**Caminhos Alternativos**
	5a	Sistema notifica o usuário que o cartão é inválido
	5b	Sistema vai para a opção de saída
Anotações		

Especificação do Caso de Uso (após verificação)

Número	1
Nome	Login no Sistema via Caixa Eletrônico
Resumo	Usuário realiza login no sistema através do caixa eletrônico
Prioridade	5
Pré-condições	Usuário possui um cartão bancário e uma senha registrados e o sistema está ativo.
Pós-condições	Usuário visualiza o menu de operações do sistema
Ator(es) Primário(s)	Cliente do Banco
Ator(es) Secundário(s)	Banco de Dados com as contas dos clientes

Gatilho	Única opção no caixa eletrônico	
Caminho Principal	**Passo**	**Ação**
	1	Sistema pede um cartão do banco
	2	Usuário insere um cartão
	3	Sistema pede uma senha
	4	Usuário insere a senha
	5	Sistema valida o cartão e a senha do usuário e mostra o menu de operações
Extensões	**Passo**	**Caminhos Alternativos**
	5a	Sistema notifica o usuário que o cartão é inválido
	5b	Sistema vai para a opção de saída, libera o cartão e retorna para o passo 1.
Anotações		

Fonte: (do autor , 2016).

5.2 Estudo Empírico

A fim de avaliar a metodologia para formalização e verificação de casos de uso, apresentada anteriormente, e seu desempenho na identificação de problemas reais e potenciais e realização de análises sobre os modelos gerados, foi realizado um estudo empírico. Como citado anteriormente, casos de uso são descritos em linguagem natural, o que pode resultar em problemas como ambiguidade e imprecisão, inerentes às linguagens naturais. Esse estudo empírico tem papel fundamental na construção de evidências concretas de que a utilização da metodologia auxilia a promover a qualidade dos casos de uso, auxiliando na identificação de problemas de especificação desses artefatos.

Para realizar uma avaliação adequada da metodologia, foram seguidos alguns princípios da engenharia de software experimental (WOHLIN ET AL., 2012). Primeiramente, foram definidos os objetivos do estudo, de acordo com o modelo GQM (BASILI, CALDIERA, ROMBACH, 1994), apresentados na Tabela 5.1.

Tabela 5.1 - Definição de objetivos do estudo, segundo o modelo GQM.

Elementos	Objetivos
Analisar	a utilidade de GGs para melhorar a qualidade de Casos de Uso
Com o propósito de	avaliar a eficiência
Com respeito a	a utilização de GGs para identificar problemas nos Casos de Uso
Da perspectiva	do pesquisador
No contexto	de um projeto simples de desenvolvimento de software

Com base nos objetivos definidos, foram derivadas duas questões de pesquisa, apresentadas a seguir, as quais serão respondidas com esse estudo.

QP-1: Os analistas são capazes de detectar problemas nas suas próprias descrições de Casos de Uso sem suporte adicional?

QP-2: Quão eficiente é a abordagem proposta, baseada em gramática de grafos, na identificação de problemas nos Casos de Uso?

A fim de responder as duas questões de pesquisa propostas, foram seguidos os passos detalhados a seguir, a qual descreve o procedimento do estudo, incluindo as métricas utilizadas para responder a questão **QP-2**. Ainda, serão apresentadas informações acerca do projeto de desenvolvimento de software utilizado no estudo.

5.2.1 Procedimentos

O procedimento do estudo consiste em quatro etapas: *Revisão dos Casos de Uso pelo Analista do Sistema*, *Formalização dos Casos de Uso*, *Avaliação dos Possíveis Problemas Encontrados* e Análise dos Dados. Cada uma das etapas é detalhada a seguir.

5.2.1.1 Revisão dos Casos de uso pelo Analista do Sistema

Com o objetivo de responder a questão **QP-1**, foi requisitado a um analista de sistemas, responsável pela criação das descrições textuais dos casos de uso para, cuidadosamente, revisá-los e pontuar os problemas encontrados, como ambiguidade, imprecisão, omissão, incompletude e inconsistência. O analista de software que colaborou com o estudo possui mais de três anos de experiência em projetos de software de diferentes complexidades, desde projetos curtos, com duração de semanas, até projetos com duração de alguns anos. Esses projetos, nos quais o analista participou, possuem documentação que descreve a arquitetura inteira da solução, incluindo artefatos como diagramas de classe, sequência e casos de uso.

Para cada problema encontrado pelo analista, é esperado que o mesmo seja detectado pelas verificações da metodologia proposta. No entanto, não há garantia de que o analista do sistema esteja apto a identificar todos os problemas existentes. Portanto, esse analista foi considerado como uma espécie de oráculo com algumas restrições, ou seja, todos os problemas identificados por ele são problemas reais, mas não necessariamente todos os problemas existentes nas descrições dos casos de uso foram detectados.

5.2.1.2 Formalização dos Casos de Uso

Inicialmente, forem selecionados cinco casos de uso para serem analisados. Após essa definição, foram realizados todos os passos detalhados anteriormente para formalizar os casos de uso utilizando o formalismo de transformações de grafos. Ainda, utilizamos a ferramenta AGG para realizar as análises propostas. Como resultado dessa etapa, foi obtido um conjunto de *Open Issues*, caracterizados como possíveis problemas, as quais destinaram-se a uma comparação com os problemas reportados pelo oráculo.

5.2.1.3 Avaliação dos Possíveis Problemas Encontrados

Após identificar possíveis problemas utilizando a abordagem baseada em transformações de grafos, foram avaliados quando esses potenciais problemas eram reais, nas descrições dos casos de uso. Se uma *Open Issue* foi apontada como problema real pelo analista, no primeiro

passo do procedimento, então esse possível problema é confirmado como real. Caso contrário, ao analista, era requisitado uma análise do possível problema encontrado e o mesmo identificava se esse potencial problema era um problema real que não foi identificado durante a inspeção manual.

5.2.1.4 Análise dos Dados

Os passos anteriores do procedimento produziram os seguintes dados: (i) uma lista de *Open Issues* identificadas; e (ii) uma lista de problemas identificados pelo analista com ou sem o auxílio da verificação proposta. O resultado ideal da verificação é que ela detecte não somente todos os problemas reais, como também que ela não detecte possíveis problemas que não são reais, ou seja, todas *Open Issues* se confirmem como problemas reais e todos os problemas sejam identificados como *Open Issues*. Isso pode ser visto como um *Problema de Classificação* e, assim, a efetividade da metodologia proposta pode ser medida utilizando as métricas amplamente conhecidas, no contexto de recuperação de informações, de *Precision* e *Recall*, cujas fórmulas são as seguintes:

$$Precision = \frac{verdadeiros\ positivos}{verdadeiros\ positivos + falsos\ positivos}$$

$$Recall = \frac{verdadeiros\ positivos}{verdadeiros\ positivos + falsos\ negativos}$$

onde os *verdadeiros positivos* são *Open Issues* que se confirmaram como problemas reais; *falsos positivos* são *Open Issues* que não se confirmaram como problemas reais; e *falsos negativos* são problemas reais não identificados como *Open Issues*.

A métrica *Precision* tem como objetivo identificar a precisão da abordagem, ou seja, a fração do conjunto de possíveis problemas que foram confirmados como reais. Já a métrica *Recall* tem como objetivo identificar a fração de elementos do conjunto de *Open Issues* que foram identificados e que são relevantes ao estudo.

5.2.2 Caso de Estudo

As descrições de Casos de Uso utilizadas nesse estudo fazem parte de uma documentação de um projeto de software industrial. Esse projeto compreende o desenvolvimento de um sistema típico de gerenciamento e venda de produtos, incluindo requisitos funcionais como adição de novos produtos, edição de informações sobre os produtos, criação de ordens de venda e gerência do estoque. Uma vez que o procedimento do

estudo envolve análise manual das descrições dos casos de uso, foram selecionados um subconjunto de todos os casos de uso disponíveis, preferencialmente aqueles contendo tarefas não triviais, com caminhos básicos e alternativos.

Os casos de uso selecionados foram escritos em inglês e descrevem ações realizados pelos atores, como usuário, sistema, base de dados, entre outros, para alcançar um estado particular do sistema ou realizar uma operação específica. Os casos de uso pertencentes ao conjunto utilizado possuem uma média de dez passos sequenciais na execução básica e cinco passos nos caminhos alternativos.

Através da estimativa realizada, baseada no *Método de Pontos por Caso de Uso* (DIEV, 2006), os casos de uso utilizados no estudo foram avaliados entre *médios* e *complexos*, devido ao número de transações, entre quatro e sete ou mais de sete, e o tipo dos atores envolvidos, em sua maioria complexos, presentes nas descrições. Devido a um acordo de confidencialidade, não é possível apresentar maiores detalhes sobre o sistema utilizado e os casos de uso selecionados para esse estudo empírico.

5.2.3 Ameaças à Validade do Estudo Empírico

Durante o planejamento e a condução do estudo, cuidadosamente foram considerados algumas preocupações relacionadas à validade do mesmo. A seguir, são discutidas as principais ameaças a validade do estudo e como as mesmas foram mitigadas.

- **Validade Interna:** A principal ameaça à validade interna desse estudo foi a seleção de uma pessoa responsável pela modelagem dos casos de uso através do formalismo de grafos. Por ser um formalismo, utilizado essencialmente dentro da subárea de métodos formais, é difícil encontrar profissionais que trabalham em projetos de software na indústria com conhecimento básico acerca de transformações de grafos. Entretanto, uma das intenções da metodologia é auxiliar esses profissionais no trabalho com grafos, para tanto, é especificado um passo a passo que, devidamente seguido, propicia a utilização de tal formalismo sem a necessidade de um conhecimento profundo sobre o mesmo. Além disso, é proposta a utilização de ferramentas, como o AGG, para automatizar as análises do modelo gerado e proporcionar manipulação dos grafos pelo usuário através de uma interface gráfica.
- **Validade da Construção:** Existem diferentes formas de modelar um sistema através do formalismo de grafos, o que pode resultar em uma ameaça quanto à validade de construção do estudo. O responsável pela modelagem pode construir

um modelo sem seguir exatamente os passos sugeridos, seguindo seu conhecimento próprio sobre o formalismo, o que resultaria em um modelo final diferente do proposto pela metodologia. Nessa situação, não há garantia de que as análises iriam obter os resultados similares aos obtidos através da metodologia. O mesmo se aplica quando o modelador possui um conhecimento avançado sobre o domínio do problema, uma vez que o modelador pode inserir informações no modelo que não estão descritas nos artefatos do software, ocultando possíveis omissões de informação na descrição dos casos de uso. Por essas razões, a metodologia apresenta um guia, passo a passo, de como modelar os casos de uso através de transformações de grafos, tanto para usuários avançados quanto iniciantes no formalismo.

- **Validade da Conclusão:** Como principal ameaça à validade de conclusão do estudo apresentado também apareceram possíveis problemas na geração do modelo através do formalismo de grafos. Além de diferentes formas de modelar um sistema e da possibilidade do modelador ser influenciado pelo seu conhecimento prévio sobre o domínio do problema, o mesmo pode construir um modelo inconsistente com a documentação, devido a erros durante o processo de modelagem. Novamente, o processo passo a passo apresentado pela metodologia deve ser seguido para prevenir que o modelo construído não reflita a descrição textual real do caso de uso. Além disso, as verificações auxiliadas por ferramentas podem detectar alguns erros de modelagem, reduzindo assim o risco dessa ameaça.
- **Validade Externa:** A principal ameaça à validade externa foi a seleção de artefatos de software sobre os quais as análises seriam realizadas. Não foram utilizados critérios para selecionar o projeto, nem o analista de sistemas que participou do estudo. Como consequência disso, o projeto utilizado pode não ser uma amostra representativa de um grande conjunto de projetos de desenvolvimento de software. Isso foi levado em consideração durante o estudo, no entanto, foi uma opção conduzir o trabalho dessa forma, escolhendo randomicamente os artefatos, a fim de mostrar a aplicabilidade da metodologia em diferentes cenários. Dessa forma, é possível garantir que os casos de uso selecionados não foram adaptados ou pertencentes a um subconjunto particular direcionado às análises realizadas. Além disso, a obtenção de bons resultados em uma situação randômica pode aumentar a confiabilidade do processo, uma vez

que não são impostos requisitos ou restrições nas descrições dos casos de uso que poderiam ser impostos pela empresa que os forneceu.

5.2.4 Resultados

Após a aplicação do procedimento descrito anteriormente, foi possível coletar os dados necessários para responder as duas questões de pesquisa definidas. O analista de sistema, após revisar os casos de uso, afirmou que eles não possuíam nenhum problema. No entanto, após a aplicação da metodologia de verificação a esses casos de uso, foram identificadas 32 *OIs* entre os cinco casos de uso, o que resulta em uma média de 6.4 *OIs* por caso de uso. Esse é um número expressivo, uma vez que o analista do sistema atestou que os casos de uso estavam corretos. A fim de verificar se as *OIs* identificadas eram reais ou alarmes falsos (falsos positivos), foi solicitado ao analista que o mesmo verificasse os possíveis problemas encontrados, um a um. Após a verificação do analista, de 32 *OIs* encontradas, 24 foram confirmadas como problemas reais e, consequentemente, somente 8 foram identificadas como falsos positivos.

A Tabela 5.2 apresenta os resultados detalhados da aplicação da metodologia em cada caso de uso. São relatados o número de *OIs* encontrados em cada caso de uso (colunas #*OI*) e quantas delas foram confirmadas como problemas reais (colunas #*P*). As linhas mostram o número de *OIs* detectados de acordo com o tipo (amarelo, laranja ou vermelho), respeitando a classificação introduzida previamente. Também é apresentado o número total de *OIs* detectados de cada tipo e o número total de problemas reais que foram identificados nos cinco casos de uso. Após a coleta dos dados, os mesmos foram analisados de acordo com as métricas selecionadas.

Tabela 5.2 - Resultados do estudo empírico.

OI	**UC 1**		**UC 2**		**UC 3**		**UC 4**		**UC 5**		**Total**	
Tipo	**#OI**	**#P**	**#OI**	**#P**	**#OI**	**#P**	**#OI**	**#P**	**#OI**	**#P**	**#OI**	**#P**
⚠	3	2	4	2	2	1	4	2	1	0	14	7
?	1	1	1	1	1	1	0	0	2	2	5	5
✋	3	3	1	1	3	2	3	3	3	3	13	12
Total	**7**	**6**	**6**	**4**	**6**	**4**	**7**	**5**	**6**	**5**	**32**	**24**

Em relação às métricas utilizadas, uma vez que o analista não foi capaz de identificar problemas sem suporte, o número de problemas não identificados pela nossa abordagem é igual a zero, levando a um *recall = 1.0*. No entanto, esse resultado pode não ser preciso, uma

vez que não é possível garantir que não existem problemas que não foram identificados nem pelo analista e nem pela verificação. Isso comprova a dificuldade de identificar problemas nos casos de uso, mesmo para *stakeholders*, sejam eles analistas ou usuários do sistema, com conhecimento sobre o domínio do problema, mostrando que não é uma tarefa trivial. Uma possível razão para isso é o fato de que esse conhecimento sobre o domínio do problema faz com que os *stakeholders* compreendam diferentes nomes como sinônimos e negligenciem possíveis omissões, entendendo que fatos não claramente definidos sejam, por vezes, óbvios. Essa situação evidencia que um suporte, seja ele através de técnicas ou ferramentas, no processo de revisão dos casos de uso tem um papel fundamental na identificação de problemas existentes.

Para a métrica *Precision* foi obtido um valor igual a *0.75*, uma vez que foram identificados 24 verdadeiros positivos e 8 falsos positivos. Isso significa que 75% das *OIs* identificadas pela metodologia proposta foram confirmados como problemas reais nas descrições dos casos de uso. Não somente a maioria dos casos foi confirmada, como também os casos que não se confirmaram como problemas reais (7 de 8) estão classificados nas categorias menos críticas.

Analisando as *OIs* não identificadas como problemas reais, foi observado que, de um total de 8 *OIs*, 6 não foram necessariamente classificadas como falsos positivos pelo analista. Nesses casos não foi feita nenhuma alteração no caso de uso, as mesmas foram postergadas para decisões em futuras fases do projeto, pois foi considerado que o analista sozinho não poderia decidir como lidar com esses possíveis problemas. As vezes a escolha feita é de se manter um certo nível de abstração na descrição de casos de uso, o que produz um risco de se ter diferentes interpretações da documentação. No entanto, esses casos foram considerados como falsos positivos porque os mesmos não foram confirmados como problemas reais. As outras 2 *OIs* encontradas, confirmadas como falsos positivos pelo analista do sistema, estão relacionadas a palavras (nomes ou conceitos) utilizadas na especificação e que foram identificadas como incompletas ou ambíguas devido a falta de conhecimento do modelador sobre o domínio do problema e os processos internos da empresa. Considerando esses resultados, é possível concluir que, na maioria dos casos, a abordagem de verificação proposta auxiliou o analista, mesmo quando as *OIs* não causaram uma alteração imediata ao caso de uso, pois pontuaram questões que podem ser consideradas em futuras fases do projeto. Essas observações podem ser utilizadas como entradas para refinamentos dos passos propostos na formalização de casos de uso utilizando transformações de grafos, a fim de criar um método assistido por ferramentas para suportar o processo de revisão de casos de uso.

É importante notar que todas as *OIs* foram identificadas sem a intervenção de nenhum *stakeholder*. O processo utiliza como entrada a documentação do software, na forma de descrições textuais dos casos de uso, e emite, como saída, uma lista de pontos a serem revisados nessa documentação. Para o analista do sistema, isso pode ser de grande valia, uma vez que os problemas detectados podem ser resolvidos, não somente nesse nível do projeto, como também nos níveis de design e implementação, uma vez que são realizados com base nos casos de uso.

5.3 Análise de Concorrência

A metodologia apresentada em Ribeiro et al. (2014) propõe a realização de um conjunto de ações, em ordem predeterminada, a fim de auxiliar na construção de um modelo formal de um Caso de Uso, através do formalismo de gramática de grafos. Após a construção, são apresentadas algumas análises que podem ser realizadas sobre o modelo, como análise do efeito do Caso de Uso e análises de pares críticos da gramática de grafos gerada, explorando os conflitos e as dependências entre as regras da gramática. Essas análises são realizadas através da ferramenta AGG, pois a mesma é capaz de computar automaticamente os relacionamentos entre as regras e gerar as informações necessárias para cada análise. Uma vez que essa ferramenta possui outros mecanismos para análises de gramáticas de grafos, além dos supracitados, não explorados pela metodologia detalhada anteriormente, são propostas, nesta subseção, algumas análises adicionais sobre o modelo formal obtido. Essas análises estão relacionadas à concorrência das ações descritas pelos Casos de Uso e são complementares às propostas pela metodologia, a fim de estender e incrementar o conjunto de análises realizadas.

A análise de concorrência de um Caso de Uso descrito em gramática de grafos é realizada através da geração de Regras Concorrentes (RC), que representam a aplicação de um conjunto de regras sobre um grafo, em uma determinada sequência. As RCs são geradas através das dependências entre regras de uma Sequência de Regras (SR) e representam a aplicação dessa SR sobre o sistema. A computação de RCs não é tarefa trivial, uma vez que envolve a identificação das dependências entre regras de uma sequência e a exploração das possíveis formas de aplicação dessas regras sobre os elementos de um grafo, o que ratifica a importância de ferramentas como a AGG no processo de análise de gramáticas de grafos. Essa análise torna possível a observação de aspectos de concorrência a partir de informações pensadas de forma sequencial, por isso mostra-se interessante no sentido de auxiliar os

desenvolvedores e *stakeholders* envolvidos a identificar possíveis problemas de concorrência antes mesmo da implementação.

Existem diferentes abordagens para se construir uma regra concorrente, uma vez que a composição de tal regra depende de quais dependências se deseja observar. As dependências são caracterizadas, nos grafos, como sobreposições de elementos. Sempre que, entre duas regras subsequentes *r1* e *r2*, existe um mesmo elemento no lado direito de *r1* e no lado esquerdo de *r2* é possível que ocorra uma sobreposição desse elemento, originando uma dependência entre as regras *r1* e *r2*, isto é, a regra *r2* consome um elemento produzido pela regra *r1*. No momento da geração de regras concorrentes é possível escolher a forma de computação a ser utilizada pela ferramenta AGG, dependendo de quais dependências se deseja observar (RUNGE, ERNEL, TAENTZER, 2012): (1) computar a RC com o máximo de sobreposições; (2) computar todas as RCs possíveis, combinando sobreposições de elementos; (3) computar a RC baseada em dependências parciais entre regras da SR; (4) computar a RC com todas as regras da SR aplicadas em paralelo, ou seja, sem sobreposições de elementos.

A primeira opção leva em consideração todas as dependências existentes entre as regras, sobrepondo o máximo de elementos possíveis em uma única RC, chamada de RC Máxima. Já a quarta opção faz exatamente o contrário, pois gera uma RC que representa a aplicação de todas as regras da SR em paralelo, ou seja, não explora as dependências entre regras, resultando em uma RC sem sobreposições.

A geração de todas as RCs possíveis, segunda opção, inclui a RC Máxima (opção 1), e os outros possíveis cenários na execução de uma SR. Por vezes a computação de todas as RCs possíveis é inviável, pois, dependendo do número de regras na SR e da cardinalidade dos elementos, definida no grafo-tipo, o número de combinações possíveis entre as regras é muito elevado, fazendo com que a ferramenta não consiga terminar a computação em tempo hábil. Uma solução que pode ser aplicada nesses casos é a definição da cardinalidade dos nós do grafo, no grafo-tipo, para no máximo 2, juntamente com a habilitação do grafo-tipo com sua cardinalidade máxima, uma vez que, no nível de abstração trabalhado, um cenário de concorrência entre 2 ou mais elementos pode ser analisado da mesma forma.

A terceira opção fornece um mecanismo para situações onde são previamente conhecidas ou se deseja estabelecer algumas dependências parciais entre as regras de uma SR, um recurso auxiliar para situações de sobrecarga no processamento das RCs. A geração das regras, nessa opção, é baseada em um conjunto de dependências parciais, definido na ferramenta através da funcionalidade *Object Flow*, a qual permite relacionar elementos de um

mesmo tipo que aparecem em regras subsequentes. Dadas duas regras subsequentes dentro de uma *SR*, *r1* e *r2*, se um mesmo nó do grafo aparece no lado direito de *r1* e no lado esquerdo e *r2*, é possível relacionar esses dois nós, definindo que, tanto o nó de *r1* quanto o nó de *r2* representam uma mesma instância de um elemento daquele tipo, criando uma dependência parcial entre as regras. Para os demais nós das regras da *SR*, quando não são definidas dependências parciais, a ferramenta considera que as operações acontecem em paralelo, sem nenhuma sobreposição de elementos entre as regras, ou seja, a ferramenta considera que o nó do lado direito da regra *r1* representa uma instância distinta daquela representada pelo nó, de mesmo tipo, que aparece no lado esquerdo da regra *r2*. Assim, é possível reduzir a complexidade da computação da RC em casos onde o não-paralelismo entre operações é conhecido ou deve ser previamente definido.

Na análise apresentada anteriormente, é explorada a geração de RCs que representam o efeito do caso de uso como um todo (passo 6.1), através da computação da RC Máxima. Nesse caso, são consideradas todas as dependências existentes entre as regras de uma SR e a RC resultante retrata uma execução sequencial do Caso de Uso, ignorando paralelismo entre regras. No entanto, as demais formas de geração de regras concorrentes fornecidas pela ferramenta AGG também podem auxiliar na identificação de possíveis problemas na descrição de um Caso de Uso, uma vez que possibilitam a análise de situações onde poderia existir paralelismo entre os passos de um Caso de Uso. A seguir, são propostas algumas análises, desenvolvidas a partir da observação de RCs e considerando as diferentes abordagens de geração das regras, exceto a primeira, já explorada na própria metodologia.

5.3.1 Extensão da Metodologia

5.3.1.1 Dependências Parciais – Object Flow

Um dos principais fatores para a análise de concorrência em um sistema é a multiplicidade dos elementos envolvidos. Isso porque, quando se define o número máximo possível de elementos de um determinado tipo em um grafo, se obtém maior clareza sobre quais operações podem ou não ser concorrentes no sistema. Quando a multiplicidade de um elemento é igual a 1, por exemplo, sabe-se que todas as regras de uma SR que envolvem tal elemento serão aplicadas sobre uma mesma instância desse elemento. No entanto, mesmo nos casos em que não se sabe ou não foi definida, previamente à implementação, a multiplicidade de algum elemento, é possível determinar quais as operações vão ocorrer sobre uma mesma

instância de um nó do grafo e, consequentemente, analisar as possibilidades de concorrência na aplicação de uma SR.

Através da funcionalidade de geração de uma RC a partir das dependências parciais é possível indicar para a ferramenta quando duas regras subsequentes devem ser aplicadas sobre uma mesma instância de um elemento. Dadas duas regras *r1* e *r2*, subsequentes em uma *SR*, e um elemento *e* do sistema (nó ou arco do grafo), se esse elemento aparece no lado direito de *r1* e no lado esquerdo de *r2*, é possível definir que essas duas regras serão aplicadas a uma mesma instância desse elemento *e*, excluindo, portanto, a situação onde existiriam dois elementos do tipo *e*, e_1 e e_2, e cada uma das regras fosse aplicada a cada um dos elementos, *r1* a e_1 e *r2* a e_2, por exemplo.

A funcionalidade para definição dessas dependências parciais é chamada *Object Flow*, ou fluxo de objeto, a qual indica quando duas regras subsequentes serão aplicadas sobre uma mesma instância de um objeto. Definindo um *Object Flow* para as regras de uma SR, é possível auxiliar a ferramenta no processamento das sobreposições de elementos entre regras, diminuindo o esforço computacional, uma vez que não são computadas situações que não estão de acordo com o *Object Flow*. É importante observar que a geração de RC através dessa funcionalidade somente considera as dependências que forem definidas, portanto as demais sobreposições que poderiam existir são ignoradas e as operações realizadas pelas regras são tratadas como ações em paralelo.

A Figura 5.6 apresenta um exemplo de definição de *Object Flow* para a SR que será utilizada como exemplo nessa seção. Essa SR é composta pelas regras *askCard*, *insertCard*, *askPwd*, *enterPwd* e *validate&display* e é referente à execução básica do Caso de Uso *Login*, apresentado anteriormente, na Figura 5.2. Nesse exemplo, é definido que a instância do objeto *Password* presente no lado direito (RHS) da regra *enterPwd* é a mesma presente no lado esquerdo (LHS) da regra *validate&display* e o mesmo se aplica ao objeto *System*. A correspondência entre as instâncias de uma regra e outra é representada, no grafo, por um número identificador do objeto, exibido ao lado do nome do mesmo, no caso o número 3 para o objeto *Password* e 1 para o objeto *System*. Para criar uma dependência parcial entre duas regras, inicialmente, é necessário selecioná-las nas listas de regras, uma na lista da esquerda (*output*) e outra na lista da direita (*input*). Quando for possível criar uma dependência entre as regras selecionadas o botão *Connect* ficará habilitado e, ao ser acionado, será exibida uma representação gráfica do lado direito da primeira regra e do lado esquerdo da segunda regra. Uma vez que as regras estiverem conectadas, é somente necessário clicar sequencialmente nos objetos do grafo que são correspondentes.

Figura 5.6 – Definição de um *Object Flow* para a Sequência de Regras que representa execução básica do Caso de Uso *Login*.

Fonte: (do autor , 2016).

A geração de RC com *Object Flow* pode ser utilizada quando se deseja definir que o efeito da aplicação de uma SR será aplicado sobre determinadas instâncias dos objetos, ou seja, em casos nos quais não há possibilidade de existência de outras instâncias de um objeto, senão as especificadas pelas dependências parciais. Portanto, quando se sabe previamente que a multiplicidade de um nó ou arco do grafo é igual a 1, isto é, só existe um objeto daquele tipo no sistema, ou quando se deseja observar o comportamento do sistema quando da aplicação de uma SR sobre uma instância específica, essa funcionalidade se apresenta como uma alternativa pertinente.

A fim de analisar o comportamento do sistema nos casos acima referidos, é possível observar algumas características da RC gerada através de um *Object Flow*, descritas abaixo na forma de *Open Issues*, seguindo a classificação proposta inicialmente pela metodologia, de acordo com a severidade da situação. Essas situações não necessariamente representam problemas reais no modelo formal, mas devem ser atentamente observadas pelo analista para posterior confirmação. De forma geral, a observação é focada nas ações que podem ser realizadas paralelamente e, por isso, podem vir a concorrer por algum recurso específico do

sistema. A análise, portanto, incide sobre a RC gerada em relação aos cenários permitidos e/ou possíveis, em busca da identificação de elementos que não estão em consonância com o esperado, ajustando-se as dependências entre regras envolvidas.

OI-C1: Existência de elementos excedentes não permitidos: A RC gerada para essa análise considera somente as dependências definidas pelo *Object Flow* e ignora as demais, portanto, após a geração, é possível identificar na RC se existem, para um tipo de nó ou arco, mais instâncias do que o permitido. A ocorrência de instâncias indesejadas pode ser causada por imprecisão na construção das regras ou pela omissão de alguma dependência no *Object Flow*. No primeiro caso, o problema pode ocorrer devido a incompletude ou inexistência de informação sobre concorrência no Caso de Uso ou, ainda, pode ser o resultado de alguma falha no processo de modelagem. Dessa forma, cada situação necessita uma análise específica para que seja proposta uma solução. Se o problema for decorrente da ausência de alguma dependência no *Object Flow*, é possível ajustar essa dependência e gerar novamente a RC, reiniciando a análise.

OI-C2. Existência de elementos excedentes possíveis: A análise da RC gerada através de um *Object Flow* possibilita, também, a identificação de nós ou arcos na RC que não estavam previstos mas que não representam uma situação inadmissível no sistema, ou seja, é possível que os mesmos apareçam, mas não era esperado que isso ocorresse. Mesmo que esses elementos sejam admissíveis na RC, a inobservância de tais situações pode implicar em um modelo com informações omissas sobre o aspecto de concorrência. Uma vez que um dos objetivos da metodologia é criar um artefato que seja capaz de auxiliar as fases futuras do projeto de software, como a implementação, é importante que se possa observar e incluir informações dessa natureza no modelo. Como as situações não representam um erro no sistema, cada caso deve ser analisado separadamente, a fim de identificar quais informações podem ser inseridas no modelo, referentes à concorrência.

5.3.1.2 Combinação de Sobreposições

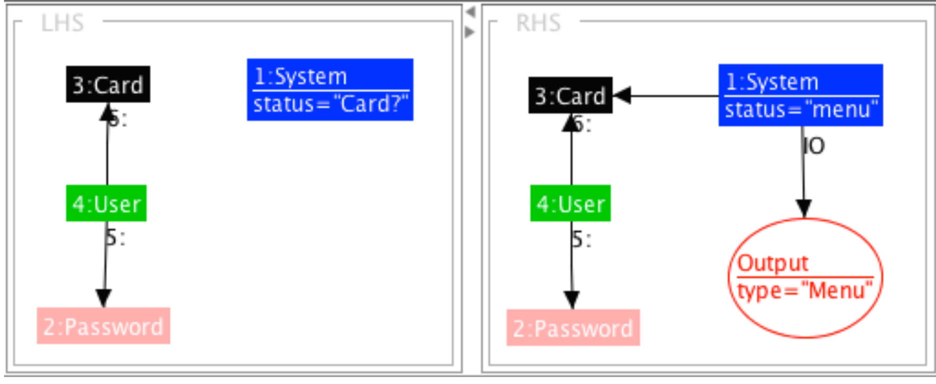
Casos de Uso são elaborados para representar operações sequenciais e, geralmente, não contêm informações sobre paralelismo e concorrência, pois na fase de criação desses artefatos nem sempre se tem definições sobre essas situações ou esses aspectos são abstraídos do modelo de Caso de Uso. No entanto, o formalismo de grafos permite a análise de situações onde existam operações concorrentes e, uma vez que exista um modelo formal em gramática

de grafos que represente esse Caso de Uso, é possível utilizar ferramentas, como o AGG, para explorar as possibilidades de execução de uma determinada SR, composta por regras dessa gramática. Através da ferramenta AGG é possível computar todas as combinações de sobreposições possíveis em uma SR, obtendo-se como resultado um conjunto de RCs que representam o comportamento do sistema em diferentes cenários de execução. Como as RCs são geradas a partir do modelo formal, construído nos passos da metodologia apresentada anteriormente, todas elas representam comportamentos válidos do sistema de acordo com o modelo.

Através da funcionalidade de geração de todas as RCs possíveis para uma SR obtêm-se todos os possíveis cenários de execução daquela SR. Portanto, uma vez que seja definida uma SR que represente os passos de uma determinada execução de um Caso de Uso, é possível gerar todas as RCs possíveis para essa execução. As RCs resultantes desse processo representam as possíveis formas de aplicação das regras daquela SR sobre um grafo, ou seja, os possíveis comportamentos do sistema em situações nas quais existem operações concorrentes. Dessa forma, a fim de continuar a análise iniciada no passo 6.1 da metodologia, é possível utilizar a mesma SR, a partir da qual foi gerada a RC Máxima, para gerar todas as possíveis RCs para aquela SR.

A geração de todas as RCs possíveis, para uma SR com n regras resulta em: uma RC Máxima, com todas as sobreposições existentes, e um conjunto de RCs contendo todas as outras possíveis combinações de sobreposições. A RC máxima, já analisada pela metodologia, será apresentada aqui para fins de comparação e análise das demais RCs. A RC Máxima para a SR que representa a execução básica do Caso de Uso *Login*, apresentado anteriormente e utilizado como exemplo, é apresentada na Figura 5.7.

Figura 5.7 – Regra Concorrente Máxima extraída da Sequência de Regras que representa execução básica do Caso de Uso *Login*.



Fonte: (do autor , 2016).

As demais RCs geradas foram classificadas em dois grupos: as *regulares* e as *eventuais*. As primeiras, simulam o paralelismo na aplicação de somente uma das regras da SR, com as demais regras da SR aplicadas sobre as mesmas instâncias dos objetos do sistema. A Figura 5.9 apresenta um exemplo de RC gerada através da combinação das dependências entre as regras da SR que representa a execução básica do Caso de Uso *Login*. As RCs regulares possuem, no seu lado esquerdo, uma união entre os elementos do lado esquerdo de uma das *n* regras da SR (L1) e os elementos da RC Máxima (L2) e, no seu lado direito, uma união entre o resultado da aplicação da regra *n* em diante sobre os elementos pertencentes ao lado esquerdo da própria regra *n* (R1) e o resultado da aplicação das regras da SR anteriores à regra *n* sobre os elementos pertencentes ao lado esquerdo da RC Máxima (R2). A regra *n*, utilizada nesse exemplo, é a regra *validate&display*, apresentada na Figura 5.8, pertencente a uma SR que contém as seguintes regras, em ordem: *askCard, insertCard, askPwd, enterPwd* e *validate&display*.

Figura 5.8 – Regra *validate&display*, referente ao passo 5 do Caso de Uso *Login*.

Fonte: (do autor , 2016).

As regras eventuais têm essa denominação porque não possuem uma generalização para sua estrutura, uma vez que podem representar execuções não previstas do Caso de Uso. Nessas RCs não há uma definição sobre onde ocorre paralelismo, pois elas são geradas através de combinações entre as outras regras, fazendo com que sejam geradas situações novas para o sistema, não necessariamente inválidas, mas que demandam atenção do analista na identificação de possíveis soluções, quando necessário. Sendo assim, para realizar a análise de concorrência a partir de todas as RCs possíveis para uma SR é necessário, após gerar as RCs, analisá-las individualmente para identificar quais eram esperadas e quais não foram previstas e, para cada tipo de RC gerada, analisar sua composição.

Figura 5.9 – Regra Concorrente gerada através de combinações entre as regras da Sequência de Regras que representa execução básica do Caso de Uso *Login*.

Fonte: (do autor , 2016).

A seguir são elencadas algumas *Open Issues*, relacionadas à composição das RCs, geradas nesta etapa. É importante lembrar que essas situações não necessariamente representam problemas reais existentes, uma vez que os problemas somente podem ser confirmados pelos *stakeholders* do projeto. Nos casos em que a geração de todas as RCs possíveis é inviável, devido ao número de combinações possíveis, a multiplicidade dos elementos pode ser definida, no grafo-tipo, com um valor máximo igual a 2 e o grafo-tipo deve ser habilitado com seu valor máximo. A análise deve ser feita sobre todas as regras geradas, com o objetivo de identificar as seguintes características:

OI-C3. Multiplicidade de nós: A multiplicidade de um nó representa o número instâncias permitidas daquele tipo de nó em um grafo, ou seja, indica quantos nós daquele tipo podem aparecer em um grafo simultaneamente. Essa multiplicidade pode ser *0, 1* ou *n* e é definida no grafo-tipo. Se em uma RC um nó aparece mais vezes do que é permitido ou desejável em um dos lados da regra, essa regra pode representar uma situação que não é válida ou que é válida e não foi analisada anteriormente. Isso porque, uma vez que tenha sido gerada uma RC com tal configuração, é possível que o sistema chegue a esse estado, de acordo com o modelo construído. Se essa situação é válida e não foi considerada quando da construção do Caso de Uso ela pode implicar em alterações na descrição do mesmo. Se a multiplicidade atual de um elemento não é

permitida, por estar incorreta ou ter sido abstraída até então, é possível ajustá-la através do grafo-tipo e, após, gerar as RCs outra vez, a fim de garantir que tal situação não ocorrerá novamente.

OI-C4. Multiplicidade de arcos: Analogamente à multiplicidade de nós, a multiplicidade de arcos representa o número instâncias permitidas daquele tipo de arco em um grafo. Essa multiplicidade também é definida no grafo-tipo e pode ter os valores *0, 1* ou *n*. Um arco representa um relacionamento existente entre os elementos que são seus vértices, portanto, se em uma RC um arco aparece mais vezes do que é permitido ou desejável em um dos lados da regra, essa regra também pode representar uma situação que não é válida ou que é válida e não foi analisada anteriormente. Da mesma forma, se essa situação é válida e não foi considerada quando da construção do Caso de Uso ela pode implicar em alterações na descrição do mesmo. Se a multiplicidade atual de um arco não é permitida, por estar incorreta ou ter sido abstraída até então, é possível ajustá-la através do grafo-tipo e, após, gerar as RCs outra vez, a fim de garantir que tal situação não ocorrerá novamente.

OI-C5. Regras eventuais: Conforme detalhado anteriormente, as RCs eventuais podem representar situações não previstas no Caso de Uso, portanto, é de grande valia a análise do efeito dessas regras, a fim de identificar se, mesmo inesperada, a configuração do sistema gerada por essas regras é válida. Caso a configuração gerada por uma RC não seja permitida no sistema, pode ser necessário garantir que o sistema não consiga chegar a tal estado, através de alterações na descrição do Caso de Uso, nas regras que compõem a SR em questão ou ainda ajustando o grafo-tipo da gramática.

É importante observar que as OIs propostas acima advém de situações inesperadas, ou até mesmo inválidas, portanto, a solução para cada caso pode requerer alterações variadas e específicas para cada caso. Além disso, todas as RCs são geradas a partir de um modelo do sistema no formalismo de grafos construído pelo analista, por essa razão não está descartada a possibilidade de que possam ser identificadas OIs decorrentes de problemas na elaboração do modelo.

5.3.2 Resultados Preliminares

Para a realização de estudos experimentais sobre o funcionamento da análise proposta, a mesma foi aplicada ao conjunto de Casos de Uso utilizado no estudo empírico apresentado anteriormente, e, após a realização da análise, foram observados os resultados expostos na Tabela 5.3, a qual apresenta detalhadamente a distribuição das OIs encontradas. As colunas da

tabela indicam os Casos de Uso e as Sequências de Regras analisadas, enquanto que as linhas indicam a quantidade de OIs identificadas de acordo com o tipo especificado previamente. Para cada Caso de Uso foram analisadas duas SRs: a primeira, SR1, corresponde a execução normal do caso de uso, ou seja, na análise do Caso de Uso *Login* apresentado na Figura 5.2, a execução básica corresponde à execução dos passos 1 a 5 do Caso de Uso, sem caminhos alternativos; enquanto que a segunda, SR2, representa a execução de algum dos caminhos alternativos, neste caso o único caminho alternativo existente, os passos 1, 2, 3, 4, 5a e 5b.

Tabela 5.3 - Resultados preliminares da análise de concorrência em Casos de Uso.

OC	**UC 1**		**UC 2**		**UC 3**		**UC 4**		**UC 5**		**Total**
OI	**SR1**	**SR2**	**SR1**	**SR2**	**SR1**	**SR2**	**SR1**	**SR2**	**SR1**	**SR2**	**OIs**
OI-C1	0	0	0	0	0	0	0	0	0	0	0
OI-C2	2	2	3	3	2	2	0	0	3	3	20
OI-C3	0	0	0	1	0	0	0	0	0	0	1
OI-C4	0	0	1	1	0	0	0	0	0	1	3
OI-C5	1	13	0	0	6	7	1	1	1	1	31
Total	**3**	**15**	**4**	**5**	**8**	**9**	**1**	**1**	**4**	**5**	**55**

A análise de concorrência, diferentemente do estudo empírico, foi realizada sem a participação de um *stakeholder* do projeto, portanto, não foi possível confirmar algumas informações, como por exemplo, se as OIs encontradas se confirmariam como problemas reais, como na análise apresentada anteriormente. Logo, em alguns casos, como na observação de OIs do tipo C1, relacionadas à existência de elementos excedentes não permitidos na RC gerada através de um *Object Flow*, não foi possível identificar OIs, uma vez que sem a participação de um responsável pelo projeto, com conhecimento sobre o domínio da aplicação, não foi possível determinar com certeza quais os elementos não poderiam aparecer na RC. Dessa forma, todos os elementos excedentes nas RCs geradas foram classificados no tipo C2, sendo tratados como elementos excedentes permitidos.

Uma OI do tipo C2 está relacionada com a existência de elementos excedentes possíveis, em uma RC. Supondo que um Caso de Uso represente a operação de *Login*, semelhante ao exposto na Figura 5.2, onde em cada passo do Caso de Uso o usuário deve entrar com uma informação, por exemplo, no primeiro o usuário insere o cartão e, no passo seguinte, uma senha. Quando, a partir desse Caso de Uso, for gerada uma Gramática de Grafos, cada passo será retratado através de uma regra. Logo, após a geração da GG, tem-se

uma regra, r_i, na qual o usuário insere no sistema seu cartão e outra regra, r_j, na qual o mesmo deverá inserir a sua senha. No entanto, as regras definem quais os tipos de elementos necessários para a sua aplicação, não suas instâncias, portanto, partindo de uma Sequência de Regras, SR_n, que contenha r_i e r_j em sequência, é possível que a Regra Concorrente gerada possua dois elementos do tipo usuário no seu lado esquerdo, u_1 e u_2 onde a aplicação da regra r_i ocorra sobre um usuário e a regra r_j seja aplicada sobre o outro usuário. Essa regra retrata a situação em que um usuário insere o seu cartão e outro usuário insere a sua senha. Tratando-se de uma operação de *login*, é provável que se deseje que as duas regras, r_i e r_j, sejam aplicadas a um mesmo usuário. Isso poderia ser definido através da criação de uma dependência, dentro da Sequência de Regras SR_n, indicando que os elementos *usuário* de r_i e de r_j são uma mesma instância. Após a criação da dependência, se a Regra Concorrente for gerada novamente, em vez de existir dois elementos do tipo *usuário*, existirá somente um.

As OIs do tipos C3 e C4, relacionadas à multiplicidade de nós e arcos nas RCs geradas, aparecem em menor número devido à sequencialidade dos Casos de Uso, expressa no modelo através das dependências entre as regras de uma SR, no entanto, incoerências quanto à multiplicidade de elementos em uma regra podem estar ligadas ao tratamento incorreto de informações. As OIs desse tipo auxiliam na identificação, por exemplo, de informações que são perdidas ou mantidas, de forma indesejada, após a aplicação de uma SR. O exemplo de OI do tipo C3 encontrado nesse estudo retrata uma situação onde, devido a impossibilidade da execução do caminho normal do Caso de Uso, uma informação foi perdida, pois o caminho alternativo não previa o tratamento dessa informação, logo, na RC gerada o elemento que representava essa informação não aparecia no lado direito da regra, mesmo com o objetivo inicial do Caso de Uso não sendo atingido, o que não necessariamente é desejável, pois o usuário poderia ter a possibilidade de acionar uma outra operação com aquela informação, ainda que a primeira tentativa tenha sido falha. As OIs do tipo C4 identificadas nesse ponto retratam situações onde ligações entre elementos deveriam ter sido excluídas, no entanto, devido a problemas como formação das regras ou falta de dependências entre elementos de uma SR, essas ligações foram mantidas, acarretando em uma RC com elementos em excesso no seu lado direito. É importante observar que, em alguns casos, a multiplicidade dos elementos foi restringida a um máximo de 2, a fim de viabilizar a computação de todas as RCs possíveis, no entanto, como a situação de paralelismo entre dois ou mais elementos é semelhante no nível de abstração trabalhado, esse resultado é igualmente aceitável.

A grande maioria das OIs encontradas pertence à categoria C5, a qual engloba todas as situações possíveis no sistema de acordo com o modelo mas que não foram previstas, como é apresentado pela tabela acima. Como, nessa etapa, a geração das RCs explora a combinação de todas as possibilidades de dependências de regras, é natural que as OIs desse tipo existam em maior quantidade, em comparação às demais. Esse tipo de OIs identifica situações que podem não ser desejáveis e que devem ser evitadas, através de algum mecanismo, como a inserção de dependências entre regras de uma SR ou de uma reestruturação das regras. É importante ressaltar que um Caso de Uso é elaborado a partir de ações sequenciais, nas quais geralmente não são considerados aspectos de concorrência. Mesmo assim, é interessante observar que, através das análises propostas, é possível extrair informações, na forma de OIs, sobre situações onde pode ocorrer paralelismo entre elementos do sistema partindo de uma fonte de informações pensada de forma sequencial.

Assim como nas análises propostas pela metodologia apresentada anteriormente, o processo proposto aqui é manual, assistido por ferramentas, e não existem garantias de que as OIs identificadas são problemas reais, uma vez que somente algum *stakeholder* do projeto seria capaz de fazer tais confirmações. No entanto, as informações extraídas a partir da análise de concorrência proporcionam a exploração desse aspecto, fundamental no desenvolvimento de softwares complexos. Além disso, a análise gera como saída um conjunto de anotações pontuais que podem servir como referência para uma posterior fase de implementação do sistema, com informações sobre concorrência e paralelismo.

5.4 Trabalhos Relacionados

Tem-se conhecimento de outros trabalhos que exploram a tradução de Diagramas de Caso de Uso para os formalismos mais difundidos, como Sistemas de Transição Rotulados (*Labelled Transitions Systems - LTS*) (SINNIG, CHALIN, KHENDEK, 2009), Redes de Petri (ZHAO, DUAN, 2009) e Máquina de Estados Finitos (*Finite State Machines - FSM*) (SINNIG, CHALIN, KHENDEK, 2013) (KLIMEK, SZWED, 2010). No entanto, esses formalismos citados são dirigidos a fluxo de controle, diferentemente do formalismo de grafos, o qual é dirigido a dados e possibilita a manipulação dos dados do sistema. Não é necessário que seja explicitamente definido o fluxo de controle, a menos que seja preciso para garantir consistência dos dados. A princípio, todas as transições especificadas (regras da gramática), na metodologia apresentada nesse capítulo, podem ser aplicadas em paralelo, desde que não causem conflito entre si, aumentando o nível de abstração do modelo. Além disso, não é determinado, na gramática de grafos, como os comportamentos devem ser

implementados, mas sim quais os comportamentos do sistema considerados corretos, o que é uma característica desejável em qualquer artefato de especificação do sistema. Ainda, existem softwares que auxiliam na manipulação de gramáticas de grafos, possibilitando a realização de refinamentos, por exemplo, a fim de aumentar a qualidade da especificação.

Considerando abordagens que formalizam Casos de Uso utilizando um modelo de transformação de grafos, existem dois trabalhos que se aproximam do apresentado nesse capítulo, apresentados por (ZIEMANN, HOLSCHER, GOGOLLA, 2005) e (HAUSMANN, HECKEL, TAENTZER, 2002), ambos utilizando também a ferramenta *AGG*. O primeiro permite a simulação da execução do sistema, mas não possibilita a realização de nenhum tipo de análises sobre o mesmo. Uma das maiores vantagens de se obter uma tradução de Casos de Uso em algum formalismo é a possibilidade de realizar análises automáticas para auxiliar na identificação de erros que dificilmente são identificados por inspeções manuais ou que não podem ser descobertos manualmente. Embora a metodologia para a tradução, validada neste trabalho, ainda não seja completamente automatizado, o processo é simples e pode ser automatizado, o que inclusive é objetivo futuro imediato dessa pesquisa.

Em (HAUSMANN, HECKEL, TAENTZER, 2002) os autores consideram análises tais como pares críticos e dependências envolvendo múltiplos Casos de Uso e apresentam algumas ideias interessantes para a interpretação dos resultados. No entanto, a metodologia apresentada neste capítulo é melhor estruturada e provê meios de se obter diagnósticos sobre o sistema, bem como serve para guiar a identificação de possíveis erros e a severidade dos mesmos. Além disso, a análise dos Casos de Uso individualmente é a verificação mais básica e fundamental que pode ser feita, para possibilitar que futuras decisões de projeto sejam tomadas de acordo com premissas confiáveis, com base nas verificações e análises já realizadas. Problemas internos de um Caso de Uso podem afetar no comportamento do mesmo em um contexto de relações com outros Casos de Uso, além de ser mais difícil identificar um problema interno quando se analisam aspectos que transcendem os limites de um único diagrama. Por essa razão, optou-se por iniciar a análise tentando melhor primeiro cada Caso de Uso em si, sendo que a análise de cenários onde existem mais de um Caso de Uso é também objetivo futuro deste trabalho, considerado como próximo passo em relação às análises já realizadas.

Embora existam abordagens similares acerca da formalização de Casos de Uso através de Gramática de Grafos, não foi encontrada nenhuma descrição de um estudo empírico, como o apresentado nesse capítulo, a fim de se validar alguma abordagem. Esse tipo de estudo é muito importante, não somente para a validação da abordagem em si, para encorajar o

desenvolvimento e a evolução do processo, mas também para auxiliar a identificar o quão eficiente é a metodologia e quais os aspectos que necessitam de melhorias. Do ponto de vista dos *stakeholders* de um projeto, os resultados obtidos aqui também são importantes, pois proporcionam aos mesmos a visualização em termos práticos, através de dados numéricos, dos benefícios da aplicação de métodos formais a um processo tipicamente informal. Além disso, a abordagem apresentada busca auxiliar os desenvolvedor a construir um modelo formal através de uma sistemática bem definida para a tradução dos Casos de Uso.

6 CONCLUSÃO

Durante o desenvolvimento do presente trabalho, buscou-se a atuação nos dois pólos do processo de desenvolvimento de software, utilizando-se tanto artefatos prévios à implementação quanto o produto final, a fim de viabilizar a construção de modelos de sistemas computacionais através do formalismo de transformação de grafos. Inicialmente, partindo de um software existente, foi proposta uma metodologia para extração de um modelo em gramática de grafos a partir de código-fonte, onde a extração de informações era baseada em execuções do sistema. Posteriormente, foi apresentada e testada uma metodologia existente para a extração de gramática de grafos a partir de diagramas de Caso de Uso, a qual foi também estendida após um estudo empírico, onde foram propostas análises sobre aspectos de concorrência das operações.

A geração de uma gramática de grafos a partir de código-fonte foi realizada a partir da instrumentação de tal código e da observação do comportamento do software durante algumas execuções do sistema, das quais foram extraídas as informações consideradas relevantes para a construção do modelo. Como resultado do processo de extração obteve-se, diretamente, um modelo formal do sistema correspondente ao código-fonte implementado, o qual pode desempenhar papel importante tanto na documentação do software, sendo incorporado à documentação existente, quanto servindo como base para manutenção e evolução do software. Além disso, através da análise desse modelo ainda é possível observar características da implementação, como propriedades ou erros no código. Após o desenvolvimento de uma abordagem inicial, foram realizados experimentos preliminares a partir dos quais foram identificadas melhorias necessárias, a fim de otimizar os resultados, as quais foram implementadas posteriormente.

A principal dificuldade encontrada no desenvolvimento dessa abordagem e na discussão de melhorias esteve relacionada à natureza dos dados utilizados para a geração da gramática. A abordagem proposta utiliza artefatos, como rastros de execução, os quais contêm, principalmente, informações dirigidas ao fluxo de controle, enquanto que o formalismo utilizado, gramática de grafos, é direcionado a dados, portanto, os modelos gerados acabaram contendo informações não naturais ao formalismo utilizado.

Embora tenham sido desenvolvidas estratégias para a redução do número de regras, o conjunto final das regras da gramática de grafos ainda possui um número elevado, dada a simplicidade dos casos de estudo utilizados. É evidente que os resultados obtidos expressam a necessidade de aperfeiçoamentos da abordagem, no entanto, tratando-se de uma abordagem

original e rudimentar, considera-se que, naturalmente, a discussão de melhorias integrará o processo de amadurecimento da mesma.

Conforme exposto no Capítulo 1, um caminho para possibilitar a verificação de um software, quanto à correção de código, é realizar a comparação entre modelos, extraídos antes e depois da implementação. Sendo assim, o modelo em Gramática de Grafos extraído nessa etapa é fundamental para possibilitar a comparação entre o que foi desejado, antes da implementação, e o que foi realmente implementado no código-fonte. Após a obtenção desse primeiro modelo em Gramática de Grafos, o estudo foi direcionado para a extração de um modelo, especificado através do mesmo formalismo, a partir de artefatos produzidos previamente à implementação, nesse caso, diagramas de Casos de Uso.

Na segunda etapa do trabalho, foi apresentada uma metodologia existente, caracterizada como um processo de tradução de casos de uso para o formalismo de transformações de grafos. A fim de testar tal metodologia e observar sua aplicabilidade, foi desenvolvido um estudo empírico sobre a eficácia da mesma na identificação de problemas em softwares, no qual foram utilizados Casos de Uso reais de uma empresa de desenvolvimento de software. Após a realização do estudo, foi possível observar que a metodologia pode produzir benefícios importantes no contexto dos projetos de desenvolvimento de software. Uma vez que sejam seguidos os passos predefinidos, juntamente com o auxílio de ferramentas, é possível, além de construir um modelo em gramática de grafos, realizar análises sobre o modelo gerado e diagnosticar reais e potenciais problemas nos casos de uso. A detecção de tais problemas pode implicar em alterações nos casos de uso e disparar uma nova rodada de análises, incremental e iterativamente, melhorando a especificação inicial. Ainda, da mesma forma que a abordagem baseada em código-fonte, a metodologia apresentada produz um modelo formal do sistema sob análise, o qual pode também ser utilizado como documentação do software e como base para análises e verificações futuras.

Fazendo uma análise geral dos resultados obtidos através do estudo empírico, é possível considerar os resultados como promissores, uma vez que foi possível, primeiramente, construir um modelo formal, ou seja, extrair uma gramática de grafos do sistema a partir dos casos de uso, além de, através das verificações propostas, melhorar a qualidade da descrição textual dos casos de uso, identificando possíveis problemas e sugerindo soluções para os mesmos. Além disso, a identificação de problemas ocorre em um estágio preliminar à implementação, o que pode implicar em uma diminuição significativa de custos.

Em relação a aplicabilidade do processo como um todo, a maior dificuldade encontra-se na automatização do processo, uma vez que o mesmo é, em sua maioria, manual. Embora faça uso de ferramentas para a realização das análises e geração das sequências de regras e das matrizes, é necessário que exista um responsável pela tradução dos casos de uso para o formalismo de grafos, através dos passos definidos por essa abordagem. Além disso, outra grande dificuldade está relacionada à portabilidade dos modelos entre ferramentas, pois as mesmas utilizam formatos distintos de arquivos para armazenar as gramáticas de grafos, impossibilitando a utilização imediata, sem uma conversão, de várias ferramentas simultaneamente, de forma complementar.

Com relação a análise de concorrência, é importante ressaltar, primeiramente, que Casos de Uso são diagramas sequenciais e, quando da elaboração desses diagramas, geralmente não é levado em consideração o aspecto de concorrência. No entanto, mesmo com essa característica, através das análises sobre o modelo de gramática de grafos, é possível identificar situações de concorrência e paralelismo, pois o formalismo fornece mecanismos para simulação e verificação de tais aspectos. Para tanto, foi proposto um conjunto de análises, baseadas na geração de regras concorrentes, funcionalidade esta oferecida pela ferramenta AGG, utilizada para auxiliar as análises. Essas análises levam em consideração os elementos que compõem as regras concorrentes, nós e arcos, bem como sua multiplicidade e possibilidades de combinações entre regras, de acordo com suas dependências.

Os resultados obtidos através da análise de concorrência se mostraram interessantes, na medida que foram identificadas diversas situações distintas de concorrência, a partir de uma documentação de software estritamente sequencial. Dessa forma, é possível estender a análise inicial, proposta pela metodologia, encorpando as verificações sobre o modelo, de forma a proporcionar uma maior cobertura do modelo aos casos de concorrência.

O ideal é que o trabalho, que atualmente é manual, de extração de informações e geração de um modelo em gramática de grafos fosse realizado automaticamente, por uma ferramenta, por exemplo, que utilizasse como entrada casos de uso ou código-fonte e, como resultado, fosse gerado o modelo. No entanto, uma vez que ainda não existem ferramentas capazes de realizar esses processos nem metodologias em estágios avançados para dirigir tais operações, é necessário que seja despendido um certo esforço em estudos investigatórios sobre técnicas e metodologias para fazer transformações entre artefatos de software e formalismos, como gramática de grafos.

Não foi possível realizar, ainda, a comparação entre os modelos gerados na primeira e segunda etapa, um dos objetivos iniciais do trabalho, a fim de possibilitar a verificação quanto à correção. Os modelos obtidos, na primeira e segunda etapa, foram criados a partir de níveis de abstração diferentes, logo, precisam ser trabalhados para que possam ser comparados. Uma primeira alternativa para essa situação é a utilização de modelos intermediários a fim de aproximar as representações, sendo esse o principal objetivo futuro desta pesquisa.

No entanto, para realizar a comparação entre modelos para fins de verificação, primeiramente, é necessário que existam mecanismos para a extração dos mesmos, que sejam capazes de descrevê-los através de uma mesma linguagem de especificação. Este trabalho apresenta-se como um passo inicial para tornar possível essa comparação, pois explora a extração de modelos em Gramática de Grafos a partir dos dois extremos do processo de desenvolvimento de software e seus artefatos. Pretende-se, ainda, evoluir a pesquisa a fim de possibilitar um maior número de verificações, porém, uma vez que o objeto de estudo são os modelos, é fundamental que se consolide uma meio confiável para a extração dos mesmos, sendo esse processo, portanto, a pedra fundamental para dar sustentação às inúmeras possibilidades de manipulação de modelos, entre elas a verificação. Além disso, mesmo sem realizar a comparação dos modelos e com um processo em estágio rudimentar, através das abordagens apresentadas neste trabalho e das análises propostas, foi possível identificar situações onde muitos aspectos, acerca do comportamento dos softwares utilizados nesta pesquisa, não foram previstos no momento das modelagens e implementações.

Ainda, a partir do desenvolvimento de pesquisas com as finalidades citadas anteriormente, será possível viabilizar a construção de ferramentas para auxiliar os desenvolvedores no processo de desenvolvimento de software, assistindo os mesmos na identificação de problemas e na melhoria dos procedimentos utilizados. Dessa forma, além de contribuir no aperfeiçoamento das atividades, a utilização de ferramentas e metodologias para verificação de sistemas através de gramática de grafos pode implicar em uma significativa redução de custos de projetos, sempre que correções decorrentes das verificações forem realizadas em fases iniciais de desenvolvimento.

REFERÊNCIAS

ALSHANQITI, A.; HECKEL, R. Towards dynamic reverse engineering visual contracts from java. **Electronic Communications of the EASST**, v. 67, 2014.

ALSHANQITI, A. M.; HECKEL, R.; KHAN, T. A. Learning minimal and maximal rules from observations of graph transformations. **ECEASST**, v. 58, 2013.

BAIER, C.; KATOEN, J. P. **Principles of Model Checking (Representation and Mind Series)**. [S.l.]: The MIT Press, 2008. ISBN 026202649X, 9780262026499.

BALL, T.; RAJAMANI, S. K. The slam project: Debugging system software via static analysis. In: ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES. Portland, OR, USA. **Proceedings...** New York: ACM, 2002. p. 1–3.

BASILI, V.; CALDIERA, C.; ROMBACH, H. **Goal Question Metric Paradigm**. [S.l.]: John Wiley & Sons, 1994. (Encyclopedia of Software Engineering, v. 1).

BÉZIVIN, J. On the unification power of models. **Software & Systems Modeling, Springer-Verlag**, v. 4, n. 2, p. 171–188, 2005. ISSN 1619-1366.

BONDY, J.-A.; MURTY, U. S. R. **Graph theory**. New York, London: Springer, 2007. (Graduate texts in mathematics). OHX. ISBN 978-1-8462-8969-9. Available from Internet: <http://opac.inria.fr/record=b1123512>.

CHAKI, S. et al. Concurrent software verification with states, events, and deadlocks. **Formal Aspects of Computing, Springer-Verlag**, v. 17, n. 4, p. 461–483, 2005.

CHIKOFSKY, E. J.; CROSSII, J. H. **Reverse engineering and de- sign recovery: A taxonomy**. IEEE Software, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 7, n. 1, p. 13–17, jan. 1990. ISSN 0740-7459.

CLARKE JR., E. M.; GRUMBERG, O.; PELED, D. A. **Model Checking**. Cambridge, MA, USA: MIT Press, 1999. ISBN 0-262-03270-8.

COMPARETTI, P. M. et al. Prospex: Protocol specification extraction. In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY, 30th, Washington, DC, USA. **Proceedings...** [S.l.]: IEEE Computer Society, 2009. p. 110–125. ISBN 978-0-7695-3633-0.

COOK, J. E.; WOLF, A. L. **Discovering models of software processes from event-based data**. v. 7, n. 3, p. 215–249, July 1998.

CORBETT, J. C. et al. Bandera: extracting finite-state models from java source code. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 22nd. New York, NY, USA. **Proceedings...** [S.l.]: ACM, 2000. p. 439–448.

CORDY, J. R. et al. Source transformation in software engineering using the {TXL} transformation system. **Information and Software Technology**, v. 44, n. 13, p. 827 – 837, 2002.

CORRADINI, A. Concurrent computing: from petri nets to graph grammars. In: **Electronic Notes in Theoretical Computer Science**. [S.l.]: Elsevier Science Publishers, 1995. p. 2.

CORRADINI, A. et al. Translating java code to graph transformation systems. In: EHRIG, H. et al. (Ed.). **Graph Transformations**. Rome, Italy: Springer Berlin Heidelberg, 2004, (Lecture Notes in Computer Science, v. 3256). p. 383–398.

DEURSEN, A. V.; VISSER, E.; WARMER, J. Model-driven software evolution: A research agenda. In: INTERNATIONAL WORKSHOP ON MODEL-DRIVEN SOFTWARE EVOLUTION HELD WITH THE ECSMR"07. **Proceedings...** [S.l.: s.n.], 2007.

DEURSEN, A. van; KLINT, P.; VISSER, J**. Domain-specific languages: An annotated bibliography**. SIGPLAN Not., ACM, New York, NY, USA, v. 35, n. 6, p. 26–36, jun. 2000. ISSN 0362-1340.

DIEV, S. Use cases modeling and software estimation: Applying use case points. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 31, n. 6, p. 1–4, nov. 2006.

DUARTE, L.; KRAMER, J.; UCHITEL, S. Towards faithful model extraction based on contexts. In: FIADEIRO, J.; INVERARDI, P. (Ed.). **Fundamental Approaches to Software Engineering**. Budapest, Hungary: Springer Berlin Heidelberg, 2008, (Lecture Notes in Computer Science, v. 4961). p. 101–115.

DUARTE, L. M. **Behaviour Model Extraction using Context Information**. Tese (Ph.D. thesis) — Imperial College London, University of London, November 2007.

DUARTE, L. M.; KRAMER, J.; UCHITEL, S. Model extraction using context information. In: NIERSTRASZ, O. et al. (Ed.). **Model Driven Engineering Languages and Systems**. Genova, Italy: Springer Berlin Heidelberg, 2006, (Lecture Notes in Computer Science, v. 4199). p. 380–394.

EHRIG, H. Introduction to the Algebraic Theory of Graph Grammars. **In: 1st Graph Grammar Workshop**. [S.l.]: Springer, 1979. (Lecture Notes in Computer Science (LNCS), v. 73), p. 1–69.

EHRIG, H. et al. (Ed.). **Handbook of graph grammars and computing by graph transformation: volume II: Applications, languages, and tools**. River Edge, USA: World Scientific, 1999.

EHRIG, H. et al. Handbook of graph grammars and computing by graph transformation. In: ROZENBERG, G. (Ed.). [S.l.]: World Scientific Publishing Co., Inc., 1997. chp. **Algebraic Approaches to Graph Transformation. Part II: Single Pushout Approach and Comparison with Double Pushout Approach**, p. 247–312.

GOMAA, H. **Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures**. [S.l.]: Cambridge University Press, 2011. ISBN 0521764149.

HAUSMANN, J. H.; HECKEL, R.; TAENTZER, G. Detection of conflicting functional requirements in a use case-driven approach: A static analysis technique based on graph

transformation. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings...** [S.l.: s.n.], 2002. p. 105–115.

HENZINGER, T. et al. Lazy Abstraction. ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES. Portland, OR, USA. **Proceedings...** [S.l.]: ACM Press, 2002. p. 58–70.

HOLZMANN, G.; SMITH, M. A practical method for verifying event-driven software. INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. Los Angeles, USA. **Proceedings...** New York: ACM, 1999. p. 597–607.

HOLZMANN, G.; SMITH, M. Software model checking - extracting verification models from source code. **In: Formal Methods for Protocol Engineering and Distributed Systems**. [S.l.]: Springer US, 1999, (IFIP Advances in Information and Communication Technology, v. 28). p. 481–497. ISBN 978-1-4757-5270-0.

JACOBSEN, E. E. **Concepts and Language Mechanisms in Software Modelling**. Dissertation (Master) — Faculty of Science and Engineering of University of Southern Denmark, Jan 2000.

KLIMEK, R.; SZWED, P. **Formal analysis of use case diagrams**. Comp. Sci., v. 11, 2010.

KORFF, M.; RIBEIRO, L. Formal relationship between graph grammars and petri nets. In: CUNY, J. et al. (Ed.). **Graph Grammars and Their Application to Computer Science**. [S.l.]: Springer Berlin Heidelberg, 1996, (Lecture Notes in Computer Science, v. 1073). p. 288–303. ISBN 978-3-540-61228-5.

LORENZOLI, D.; MARIANI, L.; PEZZE, M. Inferring state-based behavior models. In: INTERNATIONAL WORKSHOP ON DYNAMIC SYSTEMS ANALYSIS. New York, NY, USA. **Proceedings...** [S.l.]: ACM Press, 2006. p. 25–32. ISBN 1- 59593-400-6.

MACHADO, R. **Higher-order graph rewriting systems**. Tese (PhD) — Universidade Federal do Rio Grande do Sul. Instituto de Informática., Jan 2012.

MYERS, G.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. [S.l.]: Wiley, 2011. (ITPro Collection).

NIMMER, J.; ERNST, M. Automatic Generation of Program Specifications. In: INTL SYMP. ON SOFTWARE TESTING AND ANALYSIS. Rome, Italy. **Proceedings...** [S.l.:s.n.], 2002. p. 232–242.

OSBORNE, W. M.; CHIKOFSKY, E. J. Guest editors’ introduction: Fitting pieces to the maintenance puzzle. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 7, n. 1, p. 11–12, jan. 1990. ISSN 0740-7459.

PRESSMAN, R. **Engenharia de software**. [S.l.]: Makron Books, 1995. ISBN 9788534602372.

RIBEIRO, L. Métodos formais de especificação: Gramáticas de grafos. SBC Sul. VIII ESCOLA DE INFORMÁTICA DA SBC-SUL. **Anais...** [S.l.:s.n.]. 2000. p.1–33.

RIBEIRO, L. et al. Improving the quality of use cases via model construction and analysis. In: INTERNATIONAL WORKSHOP ON ALGEBRAIC DEVELOPMENT TECHNIQUES, 22nd. Sinaia, Romania. **Proceedings...** [S.l.: s.n.], 2014.

ROZENBERG, G. (Ed.). **Handbook of graph grammars and computing by graph transformations, volume 1: foundations**. [S.l.]: World Scientific Publishing Co., 1997.

RUNGE, O.; ERMEL, C.; TAENTZER, G. Agg 2.0 - new features for specifying and analyzing algebraic graph transformations. In: **Applications of Graph Transformations with Industrial Relevance.** [S.l.]: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7233). p. 81–88. ISBN 978-3-642-34175-5.

SCHMIDT, D. Guest editor's introduction: Model-driven engineering. **Computer**, v. 39, n. 2, p. 25–31, Feb 2006. ISSN 0018-9162.

SINNIG, D.; CHALIN, P.; KHENDEK, F. Lts semantics for use case models. In: ACM SYMPOSIUM ON APPLIED COMPUTING. New York, NY, USA. **Proceedings...** [S.l.]: ACM, 2009. p. 365–370. ISBN 978-1-60558-166-8.

SINNIG, D.; CHALIN, P.; KHENDEK, F. Use case and task models: An integrated development methodology and its formal foundation. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 22, n. 3, p. 27:1–27:31, jul. 2013. ISSN 1049-331X.

SOMMERVILLE, I. et al. **Engenharia de software**. [S.l.]: ADDISON WESLEY BRA, 2008. ISBN 9788588639287.

SOUZA, H. P. **Integrando modelagem intencional à modelagem de processos**. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Fev 2012.

TAENTZER, G. Agg: A tool environment for algebraic graph transformation. In: **Applications of Graph Transformations with Industrial Relevance**. [S.l.]: Springer Berlin Heidelberg, 2000, (Lecture Notes in Computer Science, v. 1779). p. 481–488. ISBN 978-3-540-67658-4.

TAENTZER, G. Agg: A graph transformation environment for modeling and validation of software. In: PFALTZ, J.; NAGL, M.; BÖHLEN, B. (Ed.). **Applications of Graph Transformations with Industrial Relevance**. Charlottesville, VA, USA: Springer Berlin Heidel- berg, 2004, (Lecture Notes in Computer Science, v. 3062). p. 446–453.

TRUYEN, F. The **Fast Guide to Model Driven Architecture - The basics of Model Driven Architecture**. [S.l.]: Cephas Consulting Group, 2006.

UTTING, M.; LEGEARD, B. **Practical Model-Based Testing: A Tools Approach**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN 0123725011, 9780080466484.

WAVEREN, R. H. van et al. **Good Modelling Practice Handbook**. Lelystad, The Netherlands: STOWA/RWS-RIZA, 2000. ISBN 90-5773-056-1.

WING, J. M. A specifier's introduction to formal methods. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 9, p. 8–23, sep. 1990. ISSN 0018-9162.

WOHLIN, C. et al. **Experimentation in Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2012. ISBN 978-3-642-29043-5.

YU, Y. et al. Reverse engineering goal models from legacy code. In: IEEE INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING, 13th, Paris, France, 2005. **Proceedings...** [S.l.: s.n.], 2005. p. 363–372.

ZHAO, C.; KONG, J.; ZHANG, K. Program behavior discovery and verification: A graph grammar approach. **IEEE Transactions on Software Engineering**, v. 36, n. 3, p. 431–448, May 2010. ISSN 0098-5589.

ZHAO, J.; DUAN, Z. Verification of use case with petri nets in requirement analysis. In: COMPUTATIONAL SCIENCE AND ITS APPLICATIONS ICCSA, 2009. **Proceedings...** [S.l.]: Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5593). ISBN 978-3-642-02456-6.

ZIEMANN, P.; HOLSCHER, K.; GOGOLLA, M. **From UML models to graph transformation systems**. ENTCS, v. 127, n. 4, p. 17 – 33, 2005.

ANEXO A – METODOLOGIA PARA FORMALIZAÇÃO DE CASOS DE USO

Use Case Analysis based on Formal Methods: An Empirical Study

Marcos Oliveira Junior, Leila Ribeiro, Érika Cota,
Lucio Mauro Duarte, Ingrid Nunes, and Filipe Reis *

PPGC - Institute of Informatics – Federal University of Rio Grande do Sul (UFRGS)
PO Box 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
{marcos.oliveira,leila,erika,lmduarte,ingridnunes,freis}@inf.ufrgs.br

Abstract. *Use Cases (UC)* are a popular way of describing system behavior and represent important artifacts for system design, analysis, and evolution. Hence, UC quality impacts the overall system quality and defect rates. However, they are presented in natural language, which is usually the cause of issues related to imprecision, ambiguity, and incompleteness. We present the results of an empirical study on the formalization of UCs as Graph Transformation models (GTs) with the goal of running tool-supported analyses on them and revealing possible errors (treated as open issues). We describe initial steps on a translation from a UC to a GT, how to use an existing tool to analyze the produced GT, and present some diagnostic feedback based on the results of these analyses and the possible level of severity of the detected problems. To evaluate the effectiveness of the translation and of the analyses in identifying problems in UCs, we applied our approach on a set of real UC descriptions obtained from a software developer company and measured the results using a well-known metric. The final results demonstrate that this approach can reveal real problems that could otherwise go undetected and, thus, help improve the quality of the UCs.
Keywords. Use Cases, Graph Transformation, Empirical Study, Model Analysis.

1 Introduction

Use Cases (UC) [3] are a popular model for documenting software expected behaviour. They are used in different software processes, not only for requirement documentation and validation, but also as specifications for system design, verification, and evolution. Hence, they are important reference points within the software development process. In current practice, UC descriptions are typically informally documented using, in most cases, natural language in a predefined structure [3]. Being informal descriptions, UCs might be ambiguous and imprecise. This may result in a number of specification problems that can be propagated to later development phases and jeopardize the overall system quality [1]. In fact, it is well-known that most software faults are introduced during the

* This work is partially supported by the VeriTeS project (FAPERGS and CNPq).

specification phase [12]. Nevertheless, it is important to keep UC descriptions in a format familiar to the stakeholders, since they must be involved in the UC definition. Thus, the verification of UCs normally corresponds to manual inspections and walkthroughs [11]. Because the analysis is manual, detecting incompleteness and recognizing ambiguities is not a trivial task. Since software quality is highly dependent on the quality of the specification, cost-effective strategies to decrease the number of errors in UCs are crucial.

Strategies for the formalization of UCs have already been proposed, such as [7], [10], [15], and [8]. Many of them assume a particular syntax for UC description tailored for their particular formalisms. This limits the expression of requirements in terms of the stakeholders language and, in some cases, also restrains the semantics of the UC. Moreover, whereas current design techniques are mostly data-driven, which delays control-flow decisions until later phases, many of the used formalisms model UCs as sequences of actions, which may neglect data-related issues. Our aim is to keep the expressiveness of a description in natural language and use a formalism for modeling/analysing UCs that is flexible enough to represent the semantics defined by stakeholders at a very abstract level. Moreover, we advocate that the translation from a UC to a formal model should be performed in a systematic way, guided by well-defined steps (possibly aided by tools), such that the model can be obtained without an expert in the formalism (because the expertise is embedded in the predefined translation process). This is fundamental for the adoption of formal methods in practice.

In this paper, we investigate the suitability of Graph Transformation (GT) [14,5] as a formal model to describe and analyze UCs. Some reasons for choosing GT are: the elements of a UC can be naturally represented as graphs; it is a visual language; the semantics is very simple yet expressive; GT is data-driven; there are various static and dynamic analysis techniques available for GT, as well as tools to support them. We work towards an approach that integrates UC formalization and tool-supported analysis, with the objective of improving the quality of UCs. As the formalization requires a precise description of the behavior described in the UC, the process of translating it into a formal model may already reveal errors. The goal is to define a sequence of steps to guide the process of building the formal model, executing analyses, and evaluating the results in terms of the level of severity of errors. Diagnostic feedback should also be provided, indicating possible actions to solve the detected problems through modification of the original UC. Hence, the process should, iteratively and gradually, improve an initial UC and generate, as result, not only a more precise UC, which can still be presented to non-technical stakeholders and be readily used without affecting the usual development process, but also a corresponding formal model that can be refined and used in subsequent design activities. This paper presents the first steps towards such a process, presenting an outline of the idea and an empirical evaluation of the effectiveness of the translation and of the analyses in identifying problems in UCs. We applied our approach on a set of real UC descriptions obtained from a software development company and measured the results using a well-known metric. The final results demonstrate

that this approach can reveal real problems that could otherwise go undetected and, thus, help improve the quality of the UCs.

This paper is organized as follows: Section 2 presents the necessary background information and details of the translation from GTs to UCs, as well as a detailed description of each step of our approach applied to a running a example; Section 3 presents the settings of the conducted empirical study; Section 4 presents an analysis and discussion of results; Section 5 discusses threats to the validity of our work; Section 6 presents a comparative analysis of our technique in relation to some similar techniques; and Section 7 concludes the paper and discusses future work.

2 Modeling UCs using GTs

2.1 Background

Use Cases According to Cockburn (2000) [3], a *Use Case (UC)* defines a contract between stakeholders of a system, describing part of the system behavior. The main purpose of a UC description is the documentation of the expected system behavior and to ease the communication between stakeholders, often including non-technical people, about required system functionalities. For this reason, the most usual UC description is the textual form. A general format of a UC contains a unique name, a primary actor, a primary goal, and a set of sequential steps describing the successful interaction between the primary actor and the system towards the primary goal. A sequence of alternative steps are often included to represent exception flows. Pre- and post-conditions are also listed to indicate, respectively, conditions that must hold before and after the UC execution.

Figure 1(a) depicts an example of UC of a bank system in a typical textual format, describing the log in operation executed by a bank client. We will explain our approach using this UC as example.

Graph Transformations The formalism of *Graph Transformations (GT)* [14,5] is based on defining states of a system as graphs and state changes as rules that transform these graphs. Due to space limitations, in this section, we will only provide an informal overview of the notions used in this paper. For formal definitions, see e.g. [14]. Examples of graphs, rules and their analysis will be given in the following subsections.

Graphs are structures that consist of a set of nodes and a set of edges. Each edge connects two nodes of the graph, one representing a source and another representing a target. A *total homomorphism* between graphs is a mapping of nodes and edges that is compatible with sources and targets of edges. Intuitively, a total homomorphism from a graph $G1$ to a graph $G2$ means that all items (nodes and edges) of $G1$ can be found in $G2$ (but distinct nodes/edges of $G1$ are not necessarily distinct in $G2$). If we have a graph, say TG, that represents all possible (graphical) types that are needed to describe a system, a total homomorphism h from any graph G to TG would associate a (graphical) type to each item of G. We call this triple $\langle G, h, TG \rangle$ a *typed graph*, and TG is called a

Use Case Specification (original)

Number	1
Name	Log into *ATM*
Summary	User logs into ATM
Priority	5
Preconditions	User has *bank card* and registered *password*
Postconditions	User receives *menu* of available *ATM* operations
Primary Actor(s)	Bank *Customer*
Secondary Actor(s)	Customer *Accounts Database*

Trigger	Only option on ATM	
Main Scenario	**Step**	**Action**
	1	*System* asks for a *Bank card*
	2	*User* inserts *card*
	3	*System* asks for *password*
	4	*User* enters *password*
	5	*System* validates user's *card* and *password* and display *menu* of operations
Extensions	**Step**	**Branching Action**
	5a	*System* notifies *user* that *password* is invalid
	5b	*System* exits option
Open Issues		

(a) Original version

Use Case Specification (after verification)

Number	1
Name	Log into System via ATM
Summary	User logs into System via ATM
Priority	5
Preconditions	User has bank card and registered password and the system is running
Postconditions	User receives menu of available System operations
Primary Actor(s)	Bank Customer
Secondary Actor(s)	Customer Accounts Database

Trigger	Only option on ATM	
Main Scenario	**Step**	**Action**
	1	System asks for a Bank card
	2	User inserts card
	3	System asks for password
	4	User enters password
	5	System validates user's card and password and display menu of operations
Extensions	**Step**	**Branching Action**
	5a	System notifies user that password is invalid
	5b	System exits option and goes back to step 1. System releases the card.
Open Issues		

(b) Improved version

Fig. 1: Login Use Case description

type graph (that is, nodes of TG describe all possible types of nodes of a system, and edges of TG describe possible relationships between these types).

A *Graph Rule* describes a relationship between two graphs. It consists of: a *left-hand side (LHS)*, which describes items that must be present for this rule to be applied; a *right-hand side (RHS)*, describing items that will be present after the application of the rule; and a *mapping from LHS to RHS*, which describes items that will be preserved by the application of the rule. This mapping must be compatible with the structure of the graphs (i.e., a morphism between typed graphs) and may be partial. Items that are in the LHS and are not mapped to the RHS are *deleted*, whereas items that are in the RHS and are not in the image of the mapping from the LHS are *created*. We also assume that rules do not merge items, that is, they are injective.

A *GT System* consists of a type graph, specifying the (graphical) types of the system, and a set of rules over this type graph that define the system behavior. The application of a rule r to a graph G is possible if an image of the LHS of r is found in G (that is, there is a total typed-graph morphism from the LHS of r to G). The result of a rule application deletes from G all items that are not mapped in r and adds the ones created by r.

Our analysis of GTs is based on concurrent rules and critical pairs, two methods of analysis independent from the initial state of the system and, thus, they are complementary to any other verification strategy based on initial states (such as testing), detailed further ahead.

2.2 UC Formalization and Verification Strategy

Figure 2 depicts the proposed UC formalization and verification strategy, which is divided into four main phases. Starting from a textual description of the UC, the first phase (*UC Data Extraction phase*) is to identify entities (Step 1) and actions (Step 2) that will be part of the formal model. Then, basic verifications can be performed regarding the consistency of the extracted information (*Primary Verifications phase*). We look for inconsistencies that might affect or even prevent the construction of the GT model such as entities or conditions that are mentioned but never used, actions or effects of an action that are not clearly defined, and so on. If inconsistencies are detected, the UC must be rewritten to eliminate them or the analyst can annotate the problem as an open issue to be resolved later on. When no basic inconsistencies are found, the GT can then be generated (*GT Generation phase*). In this process, conditions and effects of actions are modeled as states (graphs) in Step 3. Then, in Step 4, a type graph is built through the definition of a graphical representation of the artifacts generated in Steps 1 and 3. After that (Step 5), each UC step is modeled as a transition rule from one state (graph) to another, using the structures defined in Steps 3 and 4.

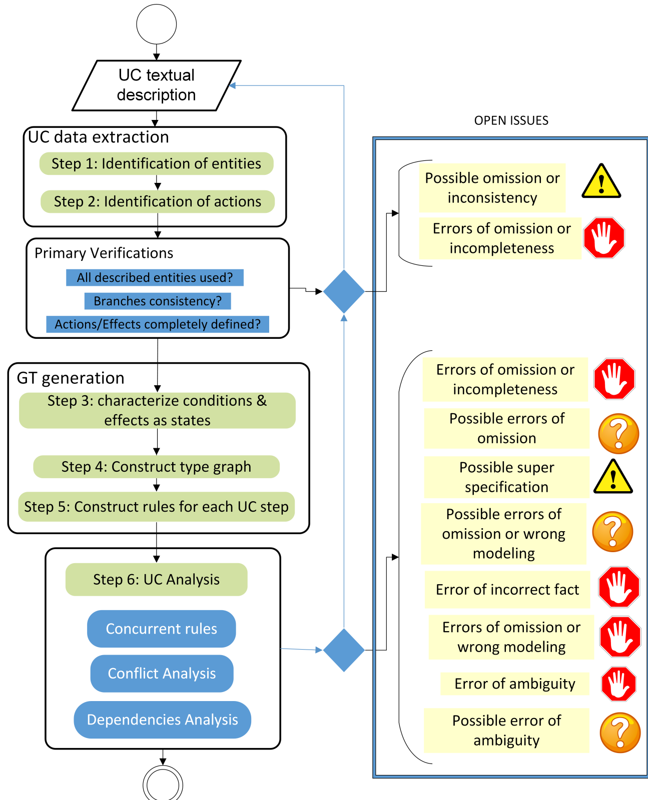


Fig. 2: Overview of the UC formalization and verification strategy.

Having the GT, a series of automatic verifications (based on concurrent rules, conflict analysis, and dependency analysis) can be performed to detect possible problems (*UC Analysis phase*). We use the AGG tool [18] to perform the automatic analyses on the GT model. All detected issues are annotated as *open issues (OIs)* along with the solutions (when applicable). With this approach, any design decision made over an OI can be documented and tracked back to the original UC. Through analysis, it is possible to verify whether the pre- and post-conditions were correctly included in the model, whether there are conflicting and/or dependent rules, what is the semantics of a detected conflict or dependency, and whether these results were expected or not. One important point is that, during the process of representing the UC in the formal model, clarifications and decisions about the semantics of the textual description must be made. Annotated OIs force the stakeholders to be more precise and explicit about tacit knowledge and unexpressed assumptions about system invariants and expected behavior.

Open issues are classified according to their severity level: code Yellow (⚠) indicates a warning, meaning a minor problem that can probably be solved by a single person; code Orange (?) indicates a problem that requires more attention and probably a definition/confirmation from the stakeholders; code Red (✋) indicates a serious issue that will require a modification in the UC description. Below, we describe the steps of our UC formalization and verification approach and the possible OIs that can be derived from them, illustrating the results for the login UC (UC1 - Fig. 1(a)).

Step 1 - *Identification of entities*: The analyst manually identifies in the UC text all the entities involved in UC. Fig. 3 shows the result for the example.

UC 1 Step 1	Artifacts	*List of Entities: User, ATM, System, Bank card, Password*

Fig. 3: UC1 - Step 1

Step 2 - *Identification of actions:* This step defines a *Table of Actions*, containing an entry for each action in the UC. Open Issues are shown in Table 1.

Open issue	**Verification**	**Problem**	**Severity level**	**Possible action**
OI.1	An entity listed in Step 1 is not used (as actor or involved) in any action	Different names for the same entity or entities used in pre-/post-conditions are not used in the steps of the UC	⚠ **Yellow**	Analyze whether this is actually what is intended,
OI.2	A branching condition is not used in any action	The description of the actions may be too abstract	⚠ **Yellow**	Analyze whether this is actually what is intended
OI.3	The effect of an action is not clearly defined	Ambiguous description or omission	✋ **Red**	Provide more details in the UC description

Table 1: Primary verification steps

Actions that perform input/output operations involve a special entity called *IO*. Considering the possibility of alternative paths described in the UC, a *Table of Branch Conditions* is also defined. With these two tables, three basic verifications can be performed and may raise open issues. Figure 4 shows the result of this step to the example UC. Only part of the Table of Actions is presented. As a result, three open issues were raised.

UC 1 Step 2

Artifacts

Table of Actions UC1

Action	Actor	Involved	Conditions	Effect
askCard	System	IO	—	Display msg asking card
insertCard	User	System, Card	1.System asks for card 2.User has card	Card becomes connected to system
...				

Table of Branch Conditions UC1

Step	Condition	Value: Step
5	User's card and password are validated	true: 5. false: 5a

Open Issues

- *ATM never appeared in the actions table* (OI.1)
- *"exits option" - step 5b - not clear* (OI.3)
- *Branching conditions was not used* : nothing was said about how validation should be carried out. (OI.2)

Fig. 4: UC1 - Step 2

Step 3 - *Modeling conditions and effects as states:* In this step, it must be explicitly defined how to describe the conditions and effects listed in the *Table of Actions*, as well as the pre- and post-conditions of the UC in terms of nodes and edges of a graph. The resulting table is called *Table of Conditions/Effects*. At the same time, we build a *Table of Operations* that is used in these formal definitions, with two predefined operations *Input* and *Output*. The tables resulting from this step are illustrated in Fig. 5.

Step 4 - *Construction of the Type graph:* The nodes of the type graph are the entities (**Step 1**) and operations (**Step 3**). The arcs are the relationships that were necessary to characterize the conditions/effects. If attributes of nodes were used to characterise the conditions/effects, they must also be part of the type graph. Figure 6 shows the type graph for our running example.

Step 5 - *Construction of rules:* Rules that formally describe the behavior of the UC are constructed. For each action listed in the *Table of Actions*, we build a rule having as left-hand side (LHS) the graph that describes the conditions that must be true for this action to occur. The graphs corresponding to each condition are already described in the *Table of Conditions/Effects*, hence it is only necessary to merge them appropriately. Analogously, the right-hand side (RHS) of the rule is built using the effects of each action. Some rules of the example UC are shown in Fig. 7.

UC 1 Step 3 — Artifacts

(Part of) Table of Conditions Effects UC1

Action	Condition/Effect	Characterization
pre	User has bank card and registered password	Password ◄— User —► Card
insertCard, askCard	System displays msg asking for card / System asks for card	System —IO→ Output type="Card?"
...		

Table of Operations UC1

OPN	Src	Tgt	RetVal	Pars	UsedIn
Output	System	—	—	type: String	askCard, askPwd, validate&display, validate¬iy
Input	—	System	—	type: String pwd: Password	enterPwd

Fig. 5: UC1 - Step 3

UC 1 Step 4 — Artifacts

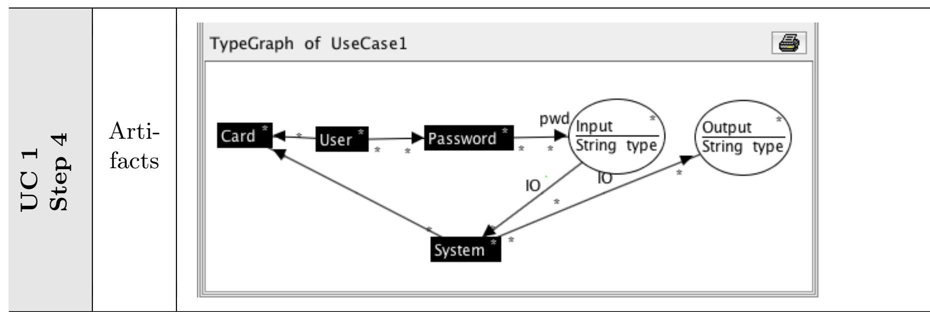


Fig. 6: UC1 - Step 4

UC 1 Step 5 — Artifacts

Set of Rules

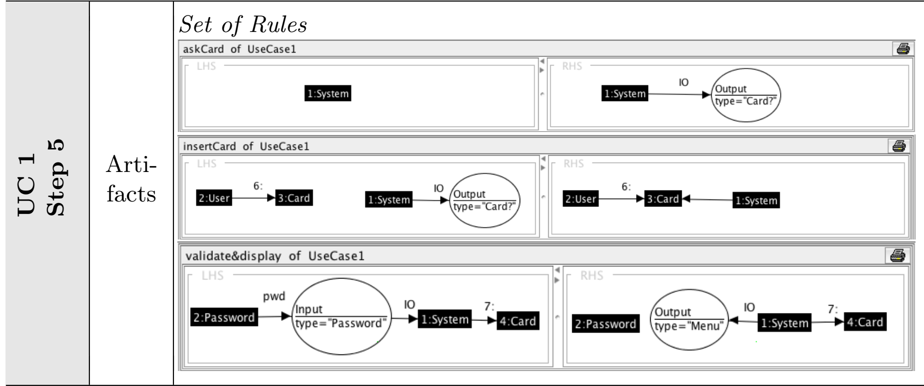


Fig. 7: UC1 - Step 5

Step 6 - *Use case analysis:* The following analysis techniques may be performed in any order. The result from these analyses are usually complementary.

6.1. Use Case Effect: We first define *rule sequences (RSs)* that represent the execution of each possible path of the UC. RSs are just sequences of rules (defined in **Step 5**) that implement the execution of each scenario of the UC. Based on each RS, we build a single rule, called *concurrent rule*, which shows the effect of the whole UC in one step. This concurrent rule allows us to check whether the overall effect is really the desired one. Table 2 presents the analysis performed on the RSs and possible resulting open issues.

Open issue	**Verification**	**Problem**	**Severity level**	**Possible action**
OI.4	A concurrent rule (for any alternative path in the UC) cannot be built using all the rules in the corresponding RSs	Items generated by some rule and used by another one may be missing by omission or modeling error	**Red**	Review the rules
OI.5	Multiple concurrent rules are built for a single UC scenario	Multiple instances of one or more entities are possible, leading to different (possibly unexpected) ways of combining the rules of the UC	**Red**	Check dependencies between rules to find unexpected subpaths in the UC behavior
OI.6	UC pre-conditions are not a subgraph of the LHSs of the concurrent rules	Pre-conditions may include unnecessary items	**Yellow**	Remove unused pre-conditions from the UC text
OI.7	The LHS of a concurrent rule is not a subgraph of the UC pre-conditions	UC requires something that is not explicitly stated in the pre-conditions	**Orange**	Identify the RS in problematic concurrent rule and check whether all actions in this path were correctly modeled. If model is correct, check for missing pre-conditions.
OI.8	Post-conditions of an alternative path of the UC are not contained in the RHS of the corresponding concurrent rule	Some rule is not generating a required item (by UC omission or modeling mistake)	**Red**	Check the rules. If all rules seem to be correct, post-conditions might be too strong.
OI.9	The RHS of a concurrent rule is not contained in the corresponding UC post-condition	Some rule is not deleting a required item (by UC omission or modeling mistake)	**Red**	If the rules seem to correctly describe each action, post-conditions might be too weak

Table 2: Verification steps on rules sequences

This analysis makes it explicit: (i) everything that is required for the UC to execute (LHS of the rule); and, (ii) the overall effect of the UC (RHS of the rule). To build the concurrent rule, the rules of the UC are joint by dependencies and, therefore, if some items are forgotten, this might lead to the impossibility of building the concurrent rule using all rules of the UC (and, thus, we might discover errors in the description of the UC steps as rules). Figure 8 shows the result of this step for our UC example.

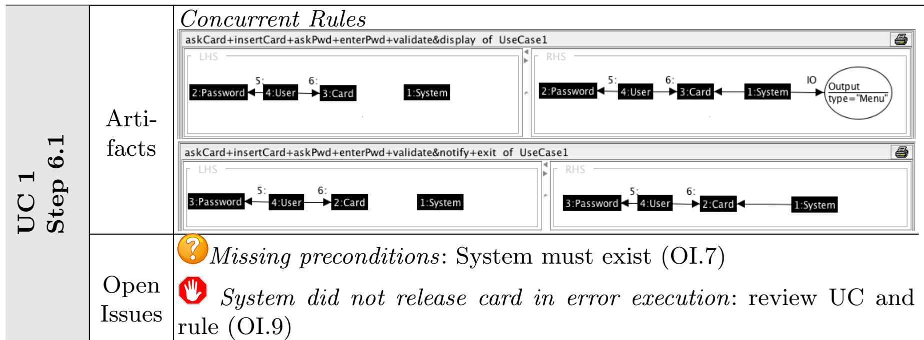


Fig. 8: UC1 - Step 6.1

6.2. Conflict Analysis (critical pairs): This type of analysis technique tells us which steps are mutually exclusive, that is, it pinpoints the choice points of the system. Table 3 presents the verifications based on critical pairs analysis and the possible resulting open issues. The result of the conflict critical-pair analysis is a *Conflict Matrix*, having rules as rows and columns, where each cell is filled with a number indicating how many items of a rule (row) are in conflict with items of another rule (column).

Open issue	**Verification**	**Problem**	**Severity level**	**Possible action**
OI.10	A rule is not conflicting with itself	The rule could be applied an arbitrary number of times	⚠ **Yellow**	Analyze whether this is actually the intended behavior
OI.11	There is no conflict between rules that represent the branching points of the UC behavior	Non-deterministic behavior: any alternative path can be taken no matter the condition	**Red**	Revise the conditions (LHSs) associated with rules representing alternative paths in the UC
OI.12	Conflicts between rules other than the ones described above (with itself and branch points)	These conflicts represent branches in system execution that must be explicitly stated in the UC (and in the model) as an alternative path	**Orange**	Revise the conflicting rules

Table 3: Verification steps based on critical pairs analysis

The conflicts are only between items of the LHSs of the rules and Zero means no conflict. The results of this step for our example are presented in Fig. 9.

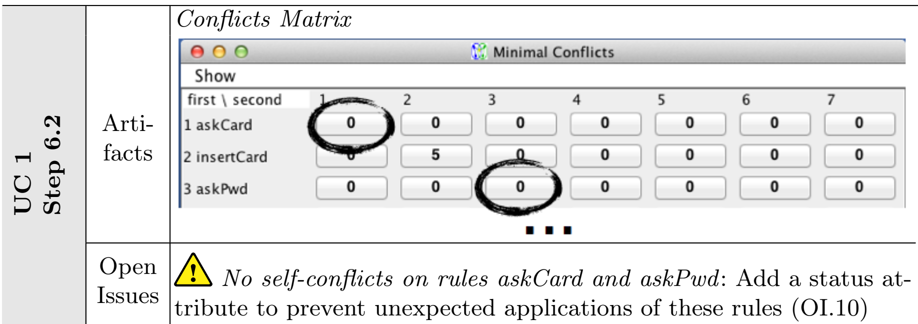
UC 1 Step 6.2		
	Artifacts	*Conflicts Matrix* 
	Open Issues	⚠ *No self-conflicts on rules askCard and askPwd*: Add a status attribute to prevent unexpected applications of these rules (OI.10)

Fig. 9: UC1 - Step 6.2

6.3. Dependency Analysis (critical pairs): Similarly to critical pair analysis, (potential) dependency analysis is independent of an initial situation and is performed by building a *Dependency Matrix*. It shows relationships between rules and can be used to check whether the dependencies that we intuitively expected to occur are actually there. The verifications based on this matrix are presented in Table 4.

Open issue	**Verification**	**Problem**	**Severity level**	**Possible action**
OI.13	Dependencies listed do not represent dependencies that are desired in the system	Possible omission in the UC description or a modeling error	⚠ **Yellow**	Check the RHS of a rule and the LHS of the other rule that depends on the first one
OI.14	An expected dependency between rules does not appear	Possible omission in the UC description or a modeling error.	⚠ **Yellow**	Check the rules involved

Table 4: Verification steps based on dependencies analysis

Figure 10 shows the result of this step for our example UC. If two rules that we would like to occur always in some specific order are shown to be independent, then they could actually occur in any order, which represents a possible problem. Hence, the rules should be checked.

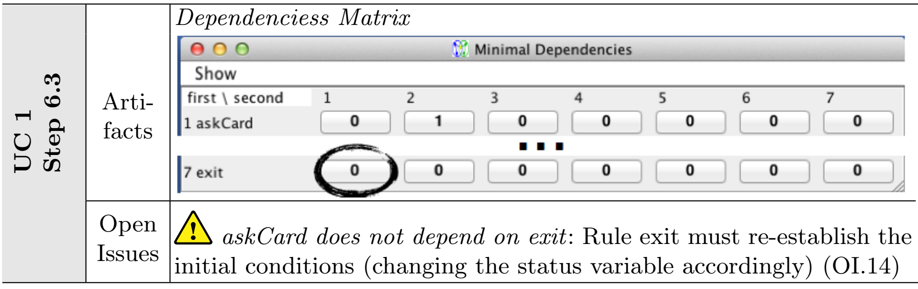
UC 1 Step 6.3		
	Artifacts	*Dependenciess Matrix* 
	Open Issues	⚠ *askCard does not depend on exit*: Rule exit must re-establish the initial conditions (changing the status variable accordingly) (OI.14)

Fig. 10: UC1 - Step 6.3

As a final result, we obtain a UC textual description more accurate and complete, as shown in Fig. 1(b), after all open issues have been analysed. Currently, most of the steps are manual and some of them are carried out aided by tools. In steps 1 and 2, the analysis is purely manual, since it is not necessary to use any tool to make the first checks. However, in steps 3, 4 and 5, the analyst should use a tool such as AGG, which helps build a formal model of the system by supporting the visual construction of graph grammars. This tool is also very useful in step 6 to perform the analysis of critical pairs and the generation of concurrent rules. Such processes are not easy to execute manually.

3 Study Settings

Considering the methodology presented in the previous section, we now detail the study conducted to evaluate its usefulness for UC formalization and its effectiveness for tool-supported analysis of UCs in order to detect real and potential problems. As already pointed out, UCs are described in natural language, which might result in problems such as ambiguity and imprecision, inherent in natural language. This empirical study was crucial to obtain concrete evidences that using GTs in the context of UCs promotes quality.

In order to adequately evaluate our approach, we followed the principles of experimental software engineering [19]. We first present our study goal in Table 5, which follows the GQM template [2].

Element	**Our study goal**
Motivation	To understand the usefulness of GTs to improve the quality of UCs
Purpose	Evaluate
Object	The effectiveness of using GTs to identify problems in UCs
Perspective	From a perspective of the researcher
Scope	In the context of a single real software development project

Table 5: Goal definition.

Based on the definition of our goal, we derived two research questions, which we aim to answer with our study. They are presented next.

RQ-1 Are system analysts able to detect problems in their own UC descriptions without additional support?

RQ-2 How effective is our GT-based approach in identifying problems in UCs?

In order to answer these research questions, we followed the steps detailed in Section 3.1, which describes our study procedure, including metrics used to answer **RQ-2**. In Section 3.2, we introduce the software development project that is the target system of our study.

3.1 Procedure

The procedure of our study consisted of the following steps:

1 - Analysis of UCs by System Analyst. In order to answer **RQ-1**, we requested a system analyst responsible for the creation of the UC descriptions, to carefully revise them, and point out problems, such as ambiguity, imprecision, omission, incompleteness, and inconsistency. This analyst has more than three years of experience in software projects with varying lengths, from a few weeks to years, with documentation describing the entire architecture of the solution, including artifacts such as class diagrams and sequence and use cases. If the system analyst found any problem, then such problems should ideally be identified by our approach. However, there was no guarantee the system analyst would be able to identify all existing problems. Therefore, the system analyst was a *"sound but not complete" oracle*, i.e. all problems identified were real problems but not necessarily all of the problems that existed in the UCs would be detected.

2 - UC Formalization. Given a set of 5 UCs, we performed the steps detailed in Section 2 to formalize them using GTs and used the AGG tool to analyze them. As a result, we detected some OIs.

3 - Evaluation of Detected Open Issues. After identifying open issues using our GT-based approach, we evaluated whether they were real problems in the analyzed UCs. If a detected OI had been pointed out as a real problem by the system analyst in the first step of our procedure, then it was definitely a real problem. Otherwise, the system analyst was requested to analyze the OI and verify whether it was an actual problem that they were unable to identify during the manual inspection.

4 - Data Analysis. The previous steps of our procedure produced the following data: (i) a list of OIs identified by our approach; and (ii) a list of problems identified by the system analyst with or without the aid of our approach. Our aim is that our approach detects all and only real problems (i.e., all OIs are real problems and all real problems are identified as OIs). This can be seen as a *classification problem*, and thus the effectiveness of our approach can be measured using the metrics widely used in the context of information retrieval of *precision* and *recall* [13], whose formulas are shown below:

$$Precision = \frac{true\ positives}{true\ positives + false\ positives} \quad (1)$$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} \quad (2)$$

where *true positives* are OIs that correspond to real problems; $false\ positives$ are OIs that are not real problems; and $false\ negatives$ are real problems not identified as OIs.

3.2 Target System

The UC descriptions we used in our study are part of the analysis documentation of an industrial software project. This project involves the development of a typical system to manage and sell products, and include functional requirements such as adding new products, changing product information, creating sale orders, and releasing products in stock. Because our study procedure involves the manual analysis of UC descriptions, we selected a subset of all available UCs, choosing those that are not trivial, involving basic and alternative flows. The selected UC descriptions were written in English and described actions performed by actors (e.g. user, system, database, etc.) to achieve a particular state of the system or perform a specific operation. The UC set used has an average of ten sequential steps and five alternative branches.

The estimate performed, based on the Use Case Points Method [4], showed that the UCs used in the study were evaluated between *average* and *complex* because of the number of transactions (between 4 and 7 or more than 7) and the type of actors involved (many complex actors) in their descriptions. We do not provide any further details about our target system and its UCs due to a confidentiality agreement.

4 Results and Discussion

As result of the execution of the procedure described in the previous section, we collected the data needed to answer our research questions. The system analyst, after revising the UCs, reported that they had no problems whatsoever. However, after applying our approach to these UCs, we identified 32 OIs across the 5 UCs, which gives an average of 6.4 OIs per UC. This is an expressive number, since the system analyst stated that the UCs were correct. In order to verify whether the identified OIs were false alarms (false positives), the system analyst was asked to check each one of them. Out of the 32 OIs, 24 were pointed out as real problems and, consequently, only 8 of the identified OIs were false positives.

Table 6 presents our results in detail. It shows the number of OIs found in each UC (columns #OI) and how many of these OIs were confirmed as real problems (columns #P). The rows show the number of detected OIs with respect to their level of severity (yellow, orange, or red), according to our previously introduced classification. The table also presents the total number of detected OIs of each type and the total number of real problems considering all the 5 UCs.

We then analyzed these results according to the selected metrics. Because the system analyst was unable to identify any problem without support, the number of problems not identified by our approach was equals to 0, leading to *recall* = 1.0. However, this result is possibly misleading, as there might be problems not identified by both the system analyst and our approach. This is an indication that even for a stakeholder (such as a system analyst, or a user) who knows well the system domain it is not a trivial task to identify problems. A possible reason for this is the fact that this knowledge of the domain causes the

Table 6: Study results

OI	UC 1		UC 2		UC 3		UC 4		UC 5		Total	
Type	**#OI**	**#P**	**#OI**	**#P**	**#OI**	**#P**	**#OI**	**#P**	**#OI**	**#P**	**#OI**	**#P**
⚠	3	2	4	2	2	1	4	2	1	0	14	7
?	1	1	1	1	1	1	0	0	2	2	5	5
✋	3	3	1	1	3	2	3	3	3	3	13	12
Total	**7**	**6**	**6**	**4**	**6**	**4**	**7**	**5**	**6**	**5**	**32**	**24**

Legend: UC - Use Case; OI - Open Issue; P - Problem.

stakeholder to understand different names as synonyms as part of this domain-specific knowledge and overlook omissions in the UCs because they believe some information is obvious. This is an evidence that support (e.g. techniques or tools) to the revision process plays a key role in identifying existing UC problems. As for the Precision, we obtained 0.75 (24 true positives and 8 false positives) — that is, 75% of the OIs identified by our GT-based approach were real problems in the UC descriptions. Not only most of the identified issues were actual problems, but also most of our errors (7 of 8) are in the less critical categories.

By analyzing OIs not identified as problems, we observed that 6 of them were not necessarily classified as a false positive by the system analyst. They preferred to leave such issues as they were and postpone changes to future design decisions, considering that they alone could not decide what was the best approach to tackle those issues. Perhaps the best approach is to maintain a certain level of abstraction in describing UCs with these issues, thus taking the risk of having different interpretations. However, we classified these issues as false positives since they were not confirmed as real problems. The other 2 OIs found, confirmed as false positives by the system analyst, were related to words (names or concepts) used in the specification and have been identified as incompleteness or ambiguities due to lack of knowledge of the modeler about the problem domain and the internal processes of the company. Considering these results, we concluded that, in most cases, our analysis helped the system analyst, even when an OI did not cause an immediate UC fix, but showed issues that might be considered in future phases of the project. These observations will be used as input for a refinement of the steps proposed to formalize UCs using GTs, in order to create a tool-assisted method to support the UC reviewing process, with increased precision in comparison to that obtained in the study described in this paper.

Note that OIs were identified without the intervention of any stakeholder. The only provided input was the software documentation in the form of UC descriptions and the output was a checklist with OIs to be revised. For the system analyst, this has great value because the detected problems can be resolved not only at the UC level, but also at the design and implementation levels, as they are performed based on UCs. More importantly, had these problems been detected before the design and implementation, when they should have, development costs could have been potentially reduced.

5 Threats to Validity

When planning and conducting our study, we carefully considered validity concerns. This section discusses the main threats we identified to validate this study and how we mitigated them.

Internal Validity. The main threat to internal validity of this study was the selection of a person responsible for performing the modeling of UCs in the formalism of graphs. Being a formalism mainly used in the sub-area of formal methods, it is difficult to find professionals working on software projects in industry with in-depth knowledge of graph transformations. However, one of our intentions with this work was to show that, correctly following the steps of our strategy, the modeler does not need a deep understanding of the formalism. Moreover, we used the AGG tool to automate the analyses of the generated model and provide a graphical interface for the manipulation of graphs.

Construct Validity. There are different ways of modeling a system through the formalism of graphs that can produce some threats to construct validity. The modeler may not follow correctly the modeling steps, being influenced by their prior knowledge about the formalism. This means that they could change the way of building the model based on their own previous knowledge. Consequently, we cannot guarantee they will obtain similar results to those presented in this work. The same applies when they have an advanced knowledge of the problem domain, because the modeler can insert information in the model that is not documented in the software artifact, hiding a possible omission of information in the UC description. For these reasons, we proposed a roadmap, step by step, on how to model UCs as GTs, for both beginners and experts users.

Conclusion Validity. As the main threat to validity of the conclusion of our study we also highlight potential problems in the generation of the model in the formalism of graphs. Besides different forms of modeling and the issue that the modeler may be influenced by their experience or prior knowledge of the problem domain, the modeler may build a model inconsistent with the initial documentation due to errors during the modeling process. Once again, our step-by-step modeling process should be followed to prevent the modeler from creating a model that is not consistent with the textual description. Moreover, the tool-supported verifications can also detect some modeling errors, as shown in Tables 2 to 4, thus reducing the risk of this threat.

External Validity. The main threat to the external validity was the selection of artifacts on which we based our study. We did not use any criteria to select either the project or the system analyst who participated of our study. As a consequence of this, the project that was made available for us may not be a representative sample of a large set of software development projects. We were aware of this threat during the study. However, we opted for randomly choosing artifacts to support the applicability of our strategy in different scenarios. This

way, we guaranteed that we were not selecting UCs that would be more tailored for our approach. We also believe that obtaining good results even in a situation of a random choice of UCs gives greater confidence on our process, as we did not impose any previous requirement or restriction on the UCs that could be provided by the company.

6 Related Work

Some authors have developed approaches for translating UCs to well-known formalisms, such as Labeled Transitions Systems [16], Petri Nets [20], and Finite State Machines [9] [17]. Unlike these formalisms, a GT model is data-driven, hence the focus is on the manipulation of data inside the system. We do not need to explicitly determine the control flow unless it is necessary to guarantee data consistency. Considering other approaches that formalize UCs using a GT model, there are two closest to ours. The approach presented in [21] allows the simulation of the execution of the system but do not report the use of any type of analysis, which, in our opinion, reduces the advantage of having a formal model. The work described in [6] considers analyses such as critical pairs and dependencies involving multiple UCs and provides some ideas on the interpretation of the results. However, we propose a more structured way of providing diagnostic feedback about single UCs, which serves as a guide to point out the possible errors as well as their severity level. As problems in individual UCs can affect the inter-UCs behavior, we chose to initially study how to improve each UC and then move on to the study of how to apply similar ideas for inter-UC formalization and analysis.

Although we could find similar approaches regarding the formalization of UCs as GT models, we could not find any description of an empirical study as the one described in this paper. We believe that this type of study is important not only to provide confidence on the proposed approach, encouraging us to develop it further in terms of its formalization as a validation and verification process, but also to allow us to quantify how good it is. This type of result is also important from the stakeholders' point of view, as they can see in practice and numerically the benefits of applying formal methods to a usually informal process. Moreover, unlike most of the other approaches, our work focuses on helping the developers to construct the formal model by the definition of a systematic translation.

7 Conclusions and Future Work

In this paper, we investigated the suitability of GT as a formal basis for UC description and improvement. We defined an outline of a translation process from UCs to GTs in a step-by-step manner, describing how to use an existing tool to analyze the generated model and diagnose real and potential problems. The detection of such problems may cause changes to the UCs and trigger a new round of analyses, incrementally and iteratively improving the initial specification. The process also generates a formal model that can be used for further

analyses. We evaluated our approach through an experiment with real software artifacts, where we could detect existing errors, which helped improve the original UCs.

Making a general analysis of the experiment, we consider the results promising, since it was possible to identify a large number of real problems based on a documentation that was produced at an early stage of a software development project. Considering the proposed strategy, we observed the need for further automation of the process, which is one of the most immediate planned future work. However, even though the process is still manual and at a low level of maturity, we had a good performance of the technique in this study considering well-known metrics, which encourages us to work on it in order to enhance the process.

In this paper, we discussed the application of our approach to one UC at a time. Even though this has already shown benefits regarding the software development process, inter-UC analyses are currently being implemented as well as the appropriate diagnostic feedback. Within the same model frame, other types of validation and verification techniques on GT models, such as test case generation, model checking, and theorem proving are also subject of current work. These techniques will be incorporated into a comprehensive methodology for software quality improvement targeting other types of errors. We also plan to study how changes in the UCs could be handled by our approach. In the current version, every modification causes a new complete round of analyses. We plan to investigate whether we could reduce the impact and cost of changes by identifying which parts are effectively affected and which analyses are actually required.

Also, a tool was designed, which is already in the early stages of development, in order to automate the first steps of the methodology (between steps 1 and 5) to help the analyst to build a formal model through the graph formalism. This tool aims to help produce the model data in a format acceptable by the AGG tool, responsible for the computational analysis of the graphs.

Finally, note that, although we did not present any new formal method or verification technique here, a considerable amount of expertise in formal methods was required to define the OIs: they are meant to bridge the gap between the informal and formal worlds. We believe that this type of work is crucial towards the industrial adoption of formal methods.

References

1. Alagar, V., Periyasamy, K.: Specification of Software Systems. Texts in Computer Science, Springer (2011)
2. Basili, V., Caldiera, C., Rombach, H.: Goal Question Metric Paradigm, Encyclopedia of Software Engineering, vol. 1. John Wiley & Sons (1994)
3. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn. (2000)
4. Diev, S.: Use cases modeling and software estimation: Applying use case points. SIGSOFT Softw. Eng. Notes 31(6), 1–4 (Nov 2006)

5. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of graph grammars and computing by graph transformation: volume II: Applications, languages, and tools. World Scientific, River Edge, USA (1999)
6. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: A static analysis technique based on graph transformation. In: Proc. of the 24th ICSE. pp. 105–115 (2002)
7. Hurlbut, R.R.: A survey of approaches for describing and formalizing use cases. Tech. Rep. XPT-TR-97-03, Expertech, Ltd. (1997)
8. Jin, N., Yang, J.: An approach of inconsistency verification of use case in XML and the model of verification tool. In: Proc. of MINES 2010. pp. 757–761 (2010)
9. Klimek, R., Szwed, P.: Formal analysis of use case diagrams. Computer Sci. 11 (2010)
10. Köters, G., werner Six, H., Winter, M.: Validation and verification of use cases and class models. In: Proc. of the 6th REFSQ (2001)
11. Myers, G., Sandler, C., Badgett, T.: The Art of Software Testing. ITPro Collection, Wiley (2011)
12. Patton, R.: Software Testing, vol. 408. Sams Publishing, 2nd edn. (2005)
13. Powers, D.M.: Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. Tech. Rep. SIE-07-001, Flinders University of South Australia (2007)
14. Rozenberg, G. (ed.): Handbook of graph grammars and computing by graph transformation: volume I: Foundations. World Scientific, River Edge, USA (1997)
15. Shen, W., Liu, S.: Formalization, testing and execution of a use case diagram. In: Dong, J., Woodcock, J. (eds.) Formal Met. and Soft. Eng., LNCS, vol. 2885, pp. 68–85. Springer Berlin Heidelberg (2003)
16. Sinnig, D., Chalin, P., Khendek, F.: LTS semantics for use case models. In: Proc. of the ACM SAC. pp. 365–370. ACM (2009)
17. Sinnig, D., Chalin, P., Khendek, F.: Use case and task models: An integrated development methodology and its formal foundation. ACM ToSEM 22(3), 27:1–27:31 (Jul 2013)
18. Taentzer, G.: AGG: A tool environment for algebraic graph transformation. In: Nagl, M., Schăźrr, A., Măźnch, M. (eds.) Applications of Graph Transformations with Industrial Relevance, LNCS, vol. 1779, pp. 481–488. Springer Berlin Heidelberg (2000)
19. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer Berlin Heidelberg (2012)
20. Zhao, J., Duan, Z.: Verification of use case with petri nets in requirement analysis. In: Proc. of the ICCSA: Part II. pp. 29–42. Springer-Verlag (2009)
21. Ziemann, P., Hăv̀lscher, K., Gogolla, M.: From UML models to graph transformation systems. ENTCS 127(4), 17 – 33 (2005)