UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

BRUNO MENEZES ROCHA

UMA ABORDAGEM DE DOIS NÍVEIS BASEADA EM VERIFICAÇÃO DE MODELOS DE UMA LÓGICA MODAL PARA AUXILIAR NA ANÁLISE DE CONFORMIDADE ARQUITETURAL DE SOFTWARE

FORTALEZA

2021

BRUNO MENEZES ROCHA

UMA ABORDAGEM DE DOIS NÍVEIS BASEADA EM VERIFICAÇÃO DE MODELOS DE UMA LÓGICA MODAL PARA AUXILIAR NA ANÁLISE DE CONFORMIDADE ARQUITETURAL DE SOFTWARE

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Ciências da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Orientadora: Prof. Dra. Ana Teresa de Castro Martins.

Coorientador: Prof. Dr. Thiago Alves Rocha.

FORTALEZA

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

R571a Rocha, Bruno Menezes.
Uma Abordagem de Dois Níveis Baseada em Verificação de Modelos de uma Lógica Modal para Auxiliar na Análise de Conformidade Arquitetural de Software / Bruno Menezes Rocha. – 2021.
117 f. : il.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2021.
Orientação: Profa. Dra. Ana Teresa de Castro Martins.
Coorientação: Prof. Dr. Thiago Alves Rocha.

1. Verificação Formal. 2. Verificação de Modelos. 3. Conformidade Arquitetural de Software. 4. Lógicas Temporais. 5. Lógicas Híbridas. I. Título.

CDD 005

BRUNO MENEZES ROCHA

UMA ABORDAGEM DE DOIS NÍVEIS BASEADA EM VERIFICAÇÃO DE MODELOS DE UMA LÓGICA MODAL PARA AUXILIAR NA ANÁLISE DE CONFORMIDADE ARQUITETURAL DE SOFTWARE

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Ciências da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Aprovada em: 29 de Outubro de 2021

BANCA EXAMINADORA

Prof. Dra. Ana Teresa de Castro Martins. (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Dr. Thiago Alves Rocha. (Coorientador)
Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE)

Prof. Dr. João Fernando Lima Alcântara
Universidade Federal do Ceará (UFC)

Prof. Dr. Prof. Dr. Lincoln Souza Rocha
Universidade Federal do Ceará (UFC)

Prof. Dr. Francicleber Martins Ferreira
Universidade Federal do Ceará (UFC)

Prof. Dr. Edward Hermann Haeusler
Pontifícia Universidade Católica (PUC)

AGRADECIMENTOS

À Professora Doutora Ana Teresa de Castro Martins, que me orientou durante a pesquisa que gerou este trabalho e durante a escrita da dissertação de mestrado.

Ao Professor Doutor Thiago Alves Rocha, coorientador deste trabalho, por suas sugestões e seus conselhos durante o desenvolvimento deste trabalho.

Ao Professor Doutor Lincoln Souza Rocha, que nos auxiliou a encontrar as direções que a pesquisa deveria tomar.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa de demanda social.

Às minhas amigas Raquel Costa Freire e Kábia R. Barbosa, que sempre me apoiam e iluminam minha vida com sua amizade.

À Taylor Swift por sua música, que me inspira na vida e que me motivou a concluir este trabalho.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) – Código de Financiamento 001.

"Sometimes giving up is the strong thing, sometimes to run is the brave thing, sometimes walking out is the one thing that will find you the right thing."

(Taylor Swift)

RESUMO

Métodos de verificação formal são essenciais para garantir que sistemas de *software* se comportem da maneira esperada. Tais métodos são utilizados na identificação de erros nos projetos dos sistemas e na comprovação que os sistemas funcionam da maneira esperada, o que aumenta a confiabilidade dos mesmos. Em Engenharia de *Software*, manter a correspondência entre a arquitetura de *software* planejada e a arquitetura que está sendo implementada é essencial para garantir a credibilidade do sistema, permitir reuso de componentes de *software* e criar sistemas que sejam fáceis de atualizar. Análise de Conformidade Arquitetural de *Software* é o processo de avaliar se a arquitetura do projeto e a arquitetura implementada estão de acordo. Neste texto, propomos uma maneira de utilizar o método de verificação formal conhecido como Verificação de Modelos para auxiliar na análise da Conformidade Arquitetural. Para formalizar as especificações, propomos uma lógica que combina conectivos de lógicas temporais, conectivos de lógicas híbridas e um operador definido neste trabalho. Para representar o sistema, usamos grafos de chamadas, que nos permitem verificar especificações de Conformidade Arquitetural relacionadas às classes e aos métodos de um sistema. Por exemplo, mostramos como formalizar especificações a respeito de padrões arquiteturais usando fórmulas da lógica que desenvolvemos com no máximo quatro conectivos temporais e simples de serem compreendidas. Grafos de versões são estruturas usadas para organizar o processo de desenvolvimento de sistemas. Aplicamos essas estruturas na verificação de requisitos globais a respeito do desenvolvimento de *software*, ou seja, especificações que contemplam várias versões. Combinando essas duas estruturas, desenvolvemos dois níveis de verificação: um nível que lida com uma versão específica de *software* e usa como base os grafos de chamadas e um nível que trata de várias versões de *software* por meio da síntese das mesmas no grafo de versões. Utilizando o operador proposto neste trabalho e uma função, conseguimos tratar esses dois níveis de maneira homogênea e elegante, utilizando uma única lógica em ambos, ao realizar a Verificação de Modelos tendo as duas estruturas citadas como base. Essa metodologia com dois níveis de verificação é a inovação deste trabalho quando comparado com outros trabalhos disponíveis na literatura de Análise de Conformidade Arquitetural: aqui, levamos em consideração a evolução do desenvolvimento do sistema. A metodologia que propomos é genérica ao funcionar como um *framework* que permite a análise de outros aspectos do sistema trocando o grafo de versões por uma estrutura adequada. Apresentamos um algoritmo de tempo polinomial para a Verificação de Modelos com a lógica

desenvolvida. A análise de complexidade do algoritmo demonstra que conseguimos conciliar expressividade e complexidade de uma maneira que pode ser aplicada na prática.

Palavras-chave: verificação formal; verificação de modelos; conformidade arquitetural de software; lógicas temporais; lógicas híbridas.

ABSTRACT

Formal methods are fundamental to ensure that software systems behave as expected. These methods are used to identify errors in system design and to certificate that systems follow the proper requirements. In Software Engineering, preserving the correspondence between the design architecture and the architecture indeed implemented by software developers is a crucial issue. Architecture Conformance Checking is the process of checking if these two architectures are in accordance along the software development course. We propose a Model Checking based method to aid Architecture Conformance Checking, which is a fundamental analysis to ensure software quality, dependability and maintainability. In this work, a new logic, which combines temporal logic, hybrid logic and a new logical operator in order to formalize software specifications, is proposed. The method described in this paper uses two structures, namely call graphs and software version graphs. The first one is used to check specifications related to classes and methods and we apply it intending to analyze a specific software version. The latter one gives us an overview of the software development process and we employ it to check global software requirements. These two graphs allow us to design a two-level checking method. The first level deals with specifications of a single software version that must be inspected in the call graph. The second level handles the global requirements throughout all software versions. Using our new operator and a function, we are able to use the same logic in both levels, allowing them to communicate with each other and handle the verification process in a neat and uniform manner. Our two-level approach is the great differential of this work, since the current approaches available in the literature focus on an unique software version at a time. We also present an algorithm, which has polynomial time complexity, to perform Model Checking for our proposed temporal logic.

Keywords: formal verification; model checking; architecture conformance checking; temporal logics; hybrid logics.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

O objetivo deste capítulo é apresentar o panorama geral deste texto. Este trabalho está inserido no contexto de Verificação Formal. Por isso falamos sobre a importância dessa área no desenvolvimento de sistemas confiáveis e de qualidade na Seção 1.1. Em seguida, apresentamos a fundamentação necessária para explanar o problema com o qual estamos lidando. Não entramos nos detalhes dos conceitos que mencionamos neste capítulo: postergamos as explicações para os capítulos seguintes. O propósito é apenas passar uma noção geral a fim de definir o problema e o objetivo deste trabalho. Isso compreende as seções 1.2 e 1.3. Elencamos as contribuições do trabalho na Seção 1.4 e apresentamos a estrutura da dissertação.

Para falar sobre Verificação Formal, usamos as referências (BAIER; KATOEN, 2008) e (RYAN; HUTH, 2004). As demais referências deste capítulo encontram-se nas seções onde são necessárias.

1.1 Verificação Formal

Sistemas de *software* estão cada vez mais presentes na sociedade. Tais sistemas permeiam a vida contemporânea: *smartphones*, carros, aparelhos médicos e televisões são alguns exemplos de objetos cotidianos altamente dependentes de *softwares*. Sistemas de *software* controlam operações críticas, como o controle dos *airbags* de um carro ou sistemas de controle de tráfego aéreo. Uma falha nesses sistemas pode levar à morte de muitas pessoas. Logo, é indispensável usar métodos formais para verificar se esses sistemas estão funcionando da maneira esperada, garantindo que as especificações do projeto estão sendo cumpridas.

À medida que sistemas computacionais ganham mais importância e são incumbidos de mais funções, a complexidade deles aumenta. Logo, torna-se cada vez mais fundamental o uso de métodos formais para assegurar que eles estão corretos, ou seja, comportam-se da maneira como foram especificados.

Verificação formal também possui um papel central em garantir que sistemas de *software* estão se comunicando de maneira adequada. Por exemplo, os carros contemporâneos possuem diferentes sistemas embutidos para controlar os freios, os *airbags*, controle de combustível, controle de direção, entre outros. Erros na comunicação entre os sistemas de um carro podem ocasionar acidentes. Por isso, é preciso garantir que as especificações determinadas estão sendo implementadas pelos *softwares* desses sistemas. Com o advento da chamada internet das

coisas, os eletrodomésticos estão cada vez mais conectados e precisam comunicar-se entre si. As comunicações ocorrem por redes cabeadas, redes sem fios e utilizando diversos protocolos. Verificação formal apresenta-se como uma ferramenta imprescindível para garantir que as comunicações estão acontecendo corretamente.

Verificação Formal é a área que utiliza métodos formais para demonstrar matematicamente que algoritmos, programas e sistemas atendem a determinadas especificações.

Modelando os sistemas por meio de máquinas de estado finito, grafos, redes de transição, semânticas formais, dentre outros formalismos matemáticos, a ideia é utilizar teoremas, especificações em lógicas, algoritmos de busca em grafos e outros artifícios matemáticos para demonstrar que os sistemas atendem a propriedades previamente especificadas e comportam-se conforme foram planejados.

Para finalizar essa seção, gostaríamos de salientar a diferença entre verificação formal e testes de *software*. Testes procuram identificar erros nos programas, porém não garantem a corretude dos *softwares* testados. Os testes consistem em executar os programas com algumas entradas, escolhidas de acordo com o tipo de teste sendo realizado, e analisar se essas produzem o resultado esperado. Se sim, não temos a garantia que o programa esteja livre de erros, porém quanto mais testes são realizados, maior é a confiabilidade do programa sendo testado. Já os métodos formais conseguem garantir a corretude de um programa. Eles funcionam como uma prova matemática, o que é muito mais confiável e preciso do que os testes.

Em sistemas mais simples, como por exemplo um aplicativo para *smartphones* com a finalidade de comprar vinhos, testes são normalmente suficientes. Entretanto, em sistemas críticos, como os já citados anteriormente ou sistemas que operam equipamentos médicos, sistemas de controle de satélites e afins, métodos formais são fundamentais para garantir que os programas funcionam em concordância com o que foi projetado.

1.2 Fundamentação teórica

Nesta seção, apresentamos as definições e os conceitos necessários para a compreensão do problema que abordamos neste trabalho. Entramos em mais detalhes sobre alguns desses conceitos nos capítulos seguintes. Falamos sobre o método de Verificação Formal que utilizamos, explanamos a área de Engenharia de *Software* onde aparece o problema que pretendemos resolver com esse método e apresentamos as estruturas que são usadas no trabalho.

1.2.1 Verificação de Modelos

Neste trabalho, utilizamos o método de Verificação Formal conhecido como Verificação de Modelos (em inglês, *Model Checking*). Verificação de Modelos (BAIER; KATOEN, 2008; RYAN; HUTH, 2004) é um método que permite analisar se um sistema satisfaz determinadas propriedades ao modelar esse sistema e então analisar os estados do modelo gerado. As propriedades desejáveis são formalizadas utilizando linguagens formais, comumente lógicas. Logo, o problema se resume a verificar se o modelo satisfaz um conjunto de fórmulas: seja M um modelo do sistema, α uma fórmula de alguma lógica que formaliza uma propriedade e s um estado de M. O objetivo da Verificação de Modelos é verificar se M satisfaz α a partir de s, o que denotamos por $M, s \models \alpha$. Dedicamos um capítulo para falar sobre esse método, a saber, o Capítulo 3.

1.2.2 Versionamento de Software

Esta seção é um compilado de informações retiradas de (SOMMERVILLE, 2011) e (CHACON; STRAUB, 2014).

Softwares são desenvolvidos incrementalmente. Adicionamos novos módulos, recursos e corrigimos *bugs* gradativamente até chegar em uma versão que atenda a todos os requisitos do *software*. Durante esse processo, é imprescindível organizar as diversas versões para facilitar o processo de gerenciamento e compreender os rumos que o sistema está tomando. Mesmo após a entrega da versão final, é importante ter um histórico organizado do desenvolvimento para realizar manutenção e atualizações do sistema. Para fazer isso, usamos versionamento de *software*.

Versionar um software significa atribuir números de versões únicos para cada estado de desenvolvimento de um *software*. Isso permite que desenvolvedores, gerentes e engenheiros saibam quando e onde foram realizadas mudanças no *software*. Os usuários podem identificar facilmente quais são as versões mais novas para manter seus sistemas atualizados.

Os números são atribuídos em ordem crescente, de modo que uma versão mais nova sempre possui um número maior do que uma versão mais antiga. É possível criar hierarquias de numeração que capturem informações relevantes para os desenvolvedores. É comum usar um padrão que consiste em níveis hierárquicos, onde incrementamos um certo nível de acordo com o grau das mudanças realizadas no *software*. Por exemplo, separando os níveis por pontos, onde os

níveis mais altos estão na frente dos níveis mais baixos, podemos ir da versão 5.1.0 para a versão 5.2.0 ao realizar a inclusão de alguma ferramenta nova no sistema. Como essa inclusão não é uma mudança grande, incrementamos o segundo nível. Mudanças menores, como a correção de um *bug*, levam ao incremento do terceiro nível, por exemplo, da versão 5.3.3 para a versão 5.3.4. Para mudanças maiores, como a adição de diversos componentes novos ao sistema, mudamos o nível mais alto e vamos da versão 5.5.6 para a 6.0.0, por exemplo. Essas decisões a respeito do impacto de mudanças cabem aos gerentes de projeto e aos desenvolvedores. Podemos nomear as versões de outros modos, por exemplo, pela data de lançamento da versão.

Um procedimento versionamento de *software* bastante conhecido e utilizado é o chamado versionamento semântico (PRESTON-WERNER,). Esse versionamento sugere que os números das versões de *software* sigam o padrão explicado a seguir.

Dado um número de versão *MAJOR.MINOR.PATCH*, incrementamos os números de acordo com o que segue:

1. Versão Maior(*MAJOR*): deve ser incrementada quando as mudanças realizadas forem incompatíveis com a API (*Application Programming Interface*) que está sendo usada pelo *software*;
2. Versão Menor(*MINOR*): deve ser incrementada quando as mudanças adicionam novas funcionalidades preservando a compatibilidade com a API usada pelo *software*;
3. Versão de Correção*(PATCH*): deve ser incrementada quando as mudanças visam corrigir falhas ou erros (*bugfix* preservando a compatibilidade com a API usada pelo *software*.

Listamos algumas outras razões que tornam o versionamento de *software* fundamental:

1. Evita confusão a respeito da versão que está sendo utilizada;
2. Facilita rastrear correções de problemas e introdução de novas funcionalidades no sistema;
3. Mantém registros de mudanças na arquitetura do *software*;
4. Permite retornar a uma versão anterior. Caso a adição de um novo recurso cause algum problema ou não saia como o desejado, podemos deletar a versão atual e voltar para a anterior;
5. Funciona não apenas para o código-fonte, mas também para *templates*, imagens e

afins relacionados à interface gráfica com o usuário;

6. Permite gestões eficientes de projetos de desenvolvimento, reduzindo tempos de desenvolvimento e economizando recursos.

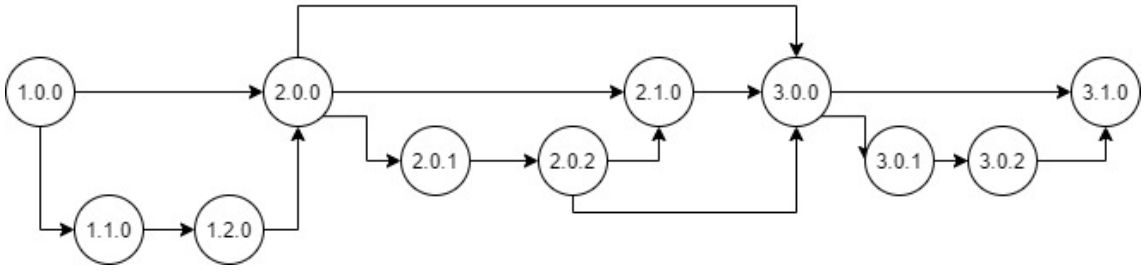
Para auxiliar no versionamento, existem Sistemas de Controle de Versões (VCS, do inglês *Version Control Systems*). Tais sistemas permitem a criação e manutenção de repositórios com os códigos-fonte, documentos e informações relevantes para o processo de desenvolvimento de *software*. É possível também manter históricos de edições e mudanças, o que é essencial para gerenciar a criação de *software*. Esses sistemas são utilizados em outras áreas e não somente na Engenharia de *Software*. Um dos exemplos mais conhecidos de VCS é o Git (CHACON; STRAUB, 2014).

Podemos estruturar o histórico mantido pelos repositórios de *software* nos chamados grafos de versões (HOFFMANN *et al.*, 2002):

Definição 1.2.1 (Grafo de versões) *Um grafo de versões é um grafo direcionado no qual os vértices representam as versões de um software e as arestas, a relação de sucessão entre as versões.*

Os grafos de versões não possuem ciclos, já que não é possível uma versão lançada em um dia x ser sucedida por uma versão lançada em um dia anterior a x. Temos apenas uma versão inicial e uma versão final. Assumimos que a partir da versão inicial, podemos alcançar todas as demais versões, já que elas são derivadas da versão inicial. Caso o leitor não esteja familiarizado com grafos, sugerimos a referência (WEST, 2000). Mostramos um exemplo ilustrativo a seguir:

Figura 1 – Exemplo de um grafo de versões



Fonte: elaborado pelo autor (2021).

Cada caminho no grafo de versões representa um ramo de desenvolvimento. Esses ramos podem ser unidos em uma versão, combinando as diferentes mudanças que foram realizadas em cada um deles, ou serem descartados. No exemplo mostrado a cima, por exemplo, temos um ramo que vai da versão 1.0.0 para a versão 2.0.0 e outro ramo que passa por duas versões intermediárias. Da versão 2.0.0 para a 3.0.0, temos quatro ramos de desenvolvimento, passando por algumas versões intermediárias. Neste caso, a versão final é a 3.1.0. Quando o *software* passar por alguma atualização ou mudança, podemos continuar a partir da versão 3.1.0 ou mesmo de alguma versão anterior, como a 3.0.0, descartando a 3.1.0.

Grafos de versões podem ser gerados pelos VCS e é comum encontrarmos esses grafos sendo chamados por outros nomes, como grafo de histórias. Esses grafos nos permitem ter uma visão global do processo de desenvolvimento. No Capítulo 4, explicamos como utilizamos grafos de versões no método de Verificação Formal proposto neste trabalho.

1.2.3 Grafos de chamadas

Os programas que constituem um *software* utilizam procedimentos, i.e, um conjunto de instruções com a finalidade de realizar uma determinada tarefa. Procedimentos chamam outros procedimentos para a execução de suas tarefas.

Em programação orientada a objetos (BRUEGGE; DUTOIT, 2009; DEITEL; DEITEL, 2014), temos a noção de classe. Uma classe é um tipo de dado definido pelos desenvolvedores. Classes são usadas para representar conceitos necessários para o desenvolvimento de *software* e possuem atributos para guardar informações e métodos associados para realizar tarefas. Métodos são apenas procedimentos atrelados a uma classe. Como exemplo, considere a classe Pessoa, apresentada a seguir em um pseudocódigo:

classe Pessoa:

Atributos:
Nome;
Sobrenome;
Idade;

Métodos:
ObterNomeCompleto();
ObterIdade();
MudarIdade();

Temos uma classe bastante simples que representa uma pessoa: adicionamos atributos para guardar o nome e a idade da pessoa e métodos que nos dizem o nome dessa pessoa e a sua idade. Temos um método que muda a idade da pessoa. No lugar disso, poderíamos adicionar um atributo data de nascimento e um método para calcular a idade da pessoa com base na data atual e na sua data de nascimento. Seja como for, as informações e os métodos que uma classe deve possuir são determinados conforme a necessidade do *software* sendo desenvolvido e o exemplo apresentado é meramente ilustrativo.

Grafos de chamadas (GROVE; CHAMBERS, 2001) são estruturas que capturam as relações de chamadas entre os diversos procedimentos de um programa:

Definição 1.2.2 (Grafo de chamadas) *Um grafo de chamadas consiste de um grafo direcionado cujos vértices representam procedimentos e as arestas, chamadas de um procedimento para outro.*

Como neste texto focamos em classes e métodos, não usaremos mais o termo procedimento: usaremos apenas o termo método. Como métodos são procedimentos, isso não representa problema algum.

Como exemplo, considere o seguinte programa escrito em Java. Trata-se de um programa que lida com o cálculo de áreas de figuras geométricas, porém não nos interessa muito os detalhes do programa, queremos apenas ilustrar um grafo de chamadas. Sublinhamos as classes com verde e os métodos envolvidos no grafo de chamadas exemplificado, com azul.

Figura 2 – Exemplo de chamadas de métodos

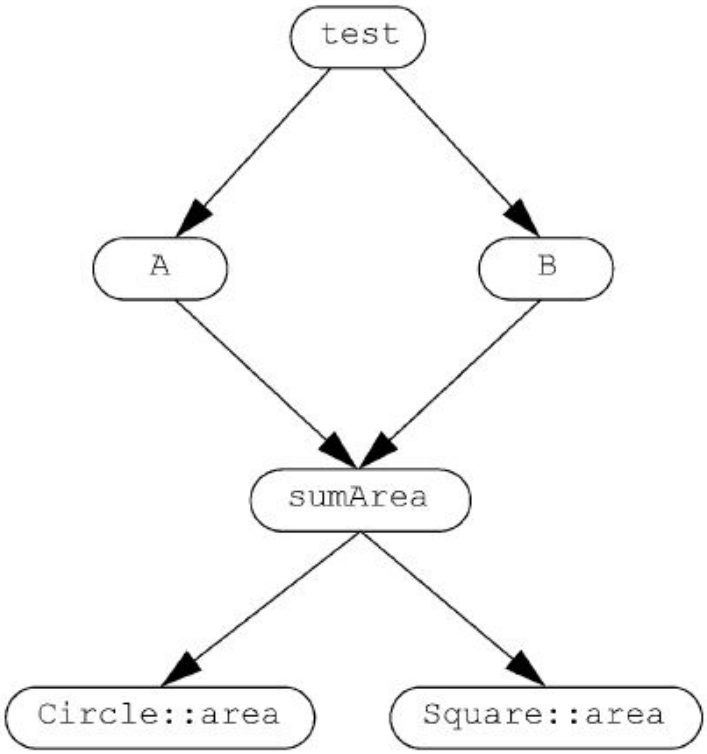
```
class Shape {
    abstract float area();
}

class Square extends Shape {
    float size;
    Square(float s) {
        size = s;
    }
    float area() {
        return size * size;
    }
}

class Circle extends Shape {
    float radius;
    Circle(float r) {
        radius = r;
    }
    float area() {
        return PI*radius*radius;
    }
}

class SPair {
    Shape first;
    Shape second;
    SPair(Shape s1, Shape s2) {
        first = s1; second = s2;
    }
}
```

```
class Example {
    float test(float v1, float v2) {
        return A(v1, v2) + B(v1, v2);
    }

    float A(float v1, float v2) {
        Circle c1 = new Circle(v1);
        Circle c2 = new Circle(v2);
        return sumArea(new SPair(c1, c2));
    }

    float B(float v1, float v2) {
        Square s1 = new Square(v1);
        Square s2 = new Square(v2);
        return sumArea(new SPair(s1, s2));
    }

    float sumArea(SPair p) {
        return p.first.area() + p.second.area();
    }
}
```

Fonte: (GROVE; CHAMBERS, 2001)

Ao invocarmos o método **test**, da classe **Example**, temos a série de chamadas de método ilustrada pelo grafo de chamadas a seguir:

Figura 3 – Grafo de chamadas



Fonte: (GROVE; CHAMBERS, 2001)

Para construir os grafos de chamadas, precisamos inspecionar o código-fonte. Percorremos o código, encontrando as chamadas entre métodos existentes no fluxo de execução que está sendo analisado e construindo o grafo de chamadas correspondente. Algoritmos para realizar essa tarefa podem ser encontrados na referência (GROVE; CHAMBERS, 2001).

Voltamos a falar sobre grafos de chamadas no Capítulo 4, apresentando os detalhes de como esses grafos são utilizados na metodologia de Verificação Formal desenvolvida neste trabalho.

1.2.4 Análise de Conformidade Arquitetural

Antes de explicar o que é Análise de Conformidade Arquitetural, precisamos explicar o que é arquitetura de *software*. A definição de arquitetura que apresentamos a seguir consiste em uma síntese das definições encontradas na literatura: (SOMMERVILLE, 2011; PERRY; WOLF, 1992; SILVA; BALASUBRAMANIAM, 2012). Apresentamos essas definições da literatura no Capítulo 2.

Definição 1.2.3 (Arquitetura de *software*) *A arquitetura de um software consiste em definir:*

1. *A especificação de seus componentes;*
2. *As propriedades desses componentes;*
3. *Os relacionamentos entre esses componentes;*
4. *As interfaces de comunicação entre os componentes e com outros softwares;*
5. *A documentação referente à arquitetura definida para o software.*

Temos inúmeros tipos de arquiteturas de *software*, conhecidos como padrões arquiteturais. Como exemplo, considere o padrão da arquitetura de repositório:

Exemplo 1 (Arquitetura de repositório) *Apresentamos uma tabela descrevendo o padrão arquitetural e, em seguida, um exemplo da aplicação desse padrão. Essa tabela deve-se a (SOMMERVILLE, 2011).*

Padrão de arquitetura de repositório	
Descrição	Todos os dados do sistema são gerenciados por um repositório central que é acessível a todos os componentes do sistema. Os componentes não interagem diretamente entre si, apenas através do repositório.
Exemplo	Apresentamos um exemplo na imagem 4.
Quando usar	Usado quando grandes volumes de informação são gerados e precisam ser armazenados por muito tempo. Pode ser usado também em sistemas dirigidos por dados, ou seja, sistemas nos quais suas atividades são compelidas por dados, onde a inclusão de novos dados aciona uma ação ou uma ferramenta.
Vantagens	Componentes são independentes: eles não precisam saber da existência de outros componentes. Mudanças feitas no repositório feitas por um componente podem ser propagadas para todos os outros componentes por meio do repositório. Os dados podem ser gerenciados consistentemente, por exemplo, *backups* são realizados ao mesmo tempo, pois todos os dados estão em apenas um lugar.
Desvantagens	O repositório é um ponto de falha único, então problemas no repositório afetam todo o sistema. A comunicação com o repositório pode ser ineficiente. Por exemplo, este pode ficar sobrecarregado se receber solicitações de muitos componentes ao mesmo tempo. Distribuir o repositório em vários computadores pode ser difícil.

Apresentamos a aplicação desse padrão no desenvolvimento de um Ambiente de Desenvolvimento Integrado, IDE, da sigla em inglês. O esquema define a estrutura do *software*:

cada caixa corresponde a um componente e as arestas definem as relações de comunicação entre esses componentes.

Figura 4 – Arquitetura de repositório

Fonte: (SOMMERVILLE, 2011)

Temos um componente central, o repositório, e uma série de componentes secundários que acessam o repositório para obter as informações armazenadas nesse repositório, como o código-fonte do projeto. Atente que esses componentes secundários não trocam informações entre si: não temos arestas entre eles. As informações são trocadas apenas com o repositório, conforme ilustrado pelas arestas direcionadas. Note que isso se encaixa na descrição que fizemos na tabela referente a esse repositório.

Quando um *software* é desenvolvido, sua arquitetura é definida pelos arquitetos de *software* e então documentada. Essa arquitetura deve ser implementada pelos desenvolvedores quando eles estão codificando o *software*. Entretanto, durante a implementação, divergências em relação ao que está no projeto do sistema aparecem. Essas divergências prejudicam a qualidade do *software* e, consequentemente, devem ser evitadas. A Análise de Conformidade Arquitetural (SILVA; BALASUBRAMANIAM, 2012; FILHO, 2016) tem como objetivo evitar tais divergências.

Manter a integridade da arquitetura planejada é fundamental para construir *softwares* de qualidade. Falamos melhor sobre esse assunto no Capítulo 2. Durante o processo de desenvolvimento, a Análise de Conformidade Arquitetural é o processo de verificar constantemente se as decisões, as definições e as especificações feitas durante a fase de projeto da arquitetura estão de fato sendo implementadas. Neste trabalho, propomos um método baseado na Verificação de

Modelos para auxiliar nessa análise.

1.2.5 Lógicas temporais e lógicas híbridas

Conforme falamos na subseção 1.2.1, no método Verificação de Modelos, precisamos de uma linguagem para formalizar as propriedades a serem verificadas. Quando esse método é aplicado na verificação de sistemas, é comum lógicas temporais serem usadas. Falamos detalhadamente sobre essas lógicas no Capítulo 3.

Lógicas temporais são lógicas modais cuja modalidade é o tempo. Essas lógicas mostraram-se apropriadas para verificar propriedades a respeito de sistemas (BAIER; KATOEN, 2008; RYAN; HUTH, 2004; FISHER, 2011), já que esses variam os seus estados ao longo do tempo. Consequentemente, necessitamos de uma lógica que consiga representar uma noção temporal para formalizar as propriedades e as especificações de interesse.

Lógicas híbridas (BLACKBURN *et al.*, 2002) utilizam um tipo especial de átomos proposicionais, chamados de nominais, para se referir aos estados de um modelo. A ideia é que cada nominal seja verdadeiro apenas em um estado do modelo. Apresentamos mais detalhes dessas lógicas no Capítulo 3. Neste trabalho, utilizamos nominais para identificar as versões de um *software* no grafo de versões do seu desenvolvimento.

No Capítulo 4, propomos uma lógica para ser utilizada na metodologia de verificação formal desenvolvida neste texto. Essa lógica utiliza operadores de lógicas temporais e lógicas híbridas, além de um novo operador proposto por este trabalho.

1.3 Objetivos e Metodologia

Nesta seção, formalizamos o problema que almejamos resolver, apresentamos o objetivo do trabalho e a metodologia que usamos para resolvê-lo.

Conforme falamos na seção anterior, desenvolvemos um método de Verificação Formal que visa auxiliar a Análise de Conformidade Arquitetural. Durante essa análise, especificações a respeito de várias partes e diversos aspectos do *software* devem ser analisadas, inclusive especificações sobre as relações de chamadas entre os métodos do *software* sendo analisado. O foco do trabalho está nessas relações entre os métodos e usamos grafos de chamadas para avaliá-las. Além disso, almejamos investigar as especificações ao longo do processo de desenvolvimento do sistema e não apenas para uma versão em particular. Para obter essa visão

global, utilizamos os grafos de versões. Propomos uma lógica, que utiliza os operadores de uma lógica temporal e de uma lógica híbrida, para a escrita formal das especificações e propriedades de *software* que devem ser verificadas.

Sumarizando o problema com o qual estamos lidando, segue que:

Dado um software e uma série de especificações sobre as relações entre os métodos desse software, verificar se o software atende ou não a tais especificações.

O objetivo desse trabalho é:

Propor um método de Verificação Formal que resolva o problema acima e permita a análise das especificações ao longo do processo de desenvolvimento de software, i.e, ao longo das suas várias versões, visando ser aplicado durante a Análise de Conformidade Arquitetural.

Com isso dito, listamos a seguir as ideias-chave da metodologia deste trabalho:

1. Desenvolver um método de Verificação Formal para a Análise de Conformidade Arquitetural com a finalidade de verificar propriedades relativas às relações entre os métodos de um *software*.
2. Analisar o comportamento dessas propriedades ao longo do processo de desenvolvimento do *software*.
3. Formalizar as propriedades de interesse usando uma lógica.
4. Aplicar Verificação de Modelos para verificar se as propriedades são satisfeitas pelo sistema. Grafos de versões e grafos de chamadas são usados para modelar o sistema e permitir que o método formal em questão seja aplicado.

Para cumprir o objetivo deste trabalho, utilizamos dois níveis de verificação:

Nível 1: tratamos de uma versão específica do *software*. Para isso, usamos os grafos de chamadas. Cada versão possui o seu grafo de chamadas. A partir desse grafo, construímos um modelo que é utilizado na Verificação de Modelos.

Nível 2: lidamos com o processo geral de desenvolvimento utilizando o grafo de versões. Esses grafos nos proporcionam uma visão global de todas as versões de um sistema. Como no nível anterior, o grafo de versões será usado para construir o modelo da Verificação de Modelos.

A conexão entre esses dois níveis é feita ao ligarmos o grafo de chamadas correspondente a uma versão ao vértice dessa versão no grafo de versões por meio de uma função. Os detalhes dessa modelagem são explicados no Capítulo 4. Na lógica que criamos, além dos operadores das lógicas temporais e híbridas, adicionamos um novo operador que permite a comunicação entre os dois níveis de verificação. Assim, conseguimos tratar esses dois níveis de maneira homogênea: atrelamos os grafos de chamadas ao grafo de versões e usamos uma única lógica, que tem um mecanismo para permitir a comunicação entre essas estruturas.

Elencamos algumas vantagens que propomos para aplicar Verificação de Modelos durante o versionamento de *software*. Analisar o comportamento da satisfatibilidade de especificações ao longo de várias versões permite coletar informações sobre como elas são e deixam de ser satisfeitas à medida que o *software* passa por mudanças. Assim, podemos analisar qual o impacto que uma mudança ou uma série de mudanças tem. Listamos as vantagens que essa análise pode trazer:

1. Facilita a depuração: informações sobre o impacto de mudanças permitem direcionar investigações da causa-raiz de um *bug* no sistema para mudanças específicas, diminuindo o tempo necessário para consertá-lo;
2. Facilita a gerência: gerentes de projeto podem administrar melhor o escopo de mudanças;
3. Analistas de requisitos podem avaliar como mudanças impactam requisitos;
4. Como as mudanças e os impactos ficam registrados, novos membros podem avaliar esse histórico para aprender sobre o projeto de *software*.

Nesta seção, apresentamos apenas panorama do que fazemos neste texto. Os detalhes são devidamente explicados nos próximos próximos capítulos. Apresentamos a estrutura do texto e apontamos as nossas contribuições na seção seguinte.

1.4 Estrutura do texto e contribuições

As contribuições deste trabalho estão listadas a seguir:

1. Propomos um método que lida de maneira homogênea com dois níveis de verificação, um que trata de uma versão particular do sistema e um nível global que captura todas as versões, permitindo analisar o comportamento das especificações e das propriedades de interesse ao longo da história de desenvolvimento de um sistema. Os métodos atuais que são usados na Análise de conformidade Arquitetural lidam apenas com uma versão por vez.
2. O método explorado neste texto utiliza o histórico de versões de *software* que é mantido por Sistemas de Controle de Versões. Assim, obtemos as vantagens desse tipo de gerenciamento, listadas na Subseção 1.2.2.
3. Apresentamos uma modelagem do sistema que liga as duas estruturas que estão sendo usadas em cada nível de verificação.
4. Propomos uma lógica temporal com um operador que permite a comunicação entre os dois níveis de verificação.
5. Uma vez que temos um modelo do sistema e as especificações formalizadas na lógica proposta, aplicamos o método Verificação de Modelos. Então, apresentamos também o algoritmo de Verificação de Modelos para a nossa metodologia. A maior parte desse algoritmo é adaptada de algoritmos de Verificação de Modelos já existentes para lógicas temporais e lógicas híbridas. Entretanto, explicamos como o algoritmo deve funcionar para o operador novo que introduzimos e para a nossa estrutura de dois níveis de verificação. Calculamos a sua complexidade.
6. A solução proposta neste trabalho é genérica: estamos usando grafos de chamadas e avaliamos propriedades relacionadas a eles. Podemos trocá-los por outras estruturas que capturem outros aspectos do *software* e aplicar a mesma metodologia descrita neste texto desde que as estruturas possam ser representadas como estruturas de Kripke, explicadas na Seção 3.2.

Este texto está organizado como segue:

1. No Capítulo 2, explicamos o que é Análise de Conformidade Arquitetural de *Software*. Explicamos o que é arquitetura de *software*, conceito fundamental para entender a importância da Conformidade Arquitetural, apresentamos o problema que a Análise de Conformidade

visa solucionar e fazemos um levantamento sobre as abordagens e métodos existentes para realizar essa análise. Ao mostrar a necessidade da Análise de Conformidade Arquitetural, esse capítulo serve como a motivação para a metodologia que propomos neste trabalho.

2. No Capítulo 3, apresentamos o método de Verificação de Modelos. O objetivo desse capítulo é apresentar a fundamentação teórica na qual se baseia a metodologia desenvolvida neste texto. Por isso, falamos sobre as lógicas que adaptamos para criar a lógica que propomos e explicamos os algoritmos de Verificação de Modelos para essas lógicas.
3. No Capítulo 4, explicamos a metodologia que propomos neste trabalho. Apresentamos as definições de grafos de versões e de grafos de chamadas e explicamos como modelar o sistema a partir dessas estruturas com a finalidade de aplicarmos a Verificação de Modelos. Explicamos a sintaxe e a semântica da lógica que estamos propondo, apresentamos exemplos de especificações e como elas são escritas nessa lógica. Explanamos o funcionamento do algoritmo de Verificação de Modelos para essa lógica e calculamos a sua complexidade.
4. No Capítulo 5, fazemos um estudo de caso para mostrar o funcionamento da metodologia no desenvolvimento de um programa simples.
5. No Capítulo 6, apresentamos a conclusão desse trabalho e finalizamos com propostas de trabalhos futuros. Mostramos como podemos alterar a metodologia que estamos propondo para verificar especificações que falam sobre outros aspectos do *software*, já que o foco do trabalho é nas especificações a respeito das relações entre os métodos, e apontamos melhorias que podem ser feitas na metodologia apresentada.

2 CONFORMIDADE ARQUITETURAL DE *SOFTWARE*

O objetivo deste capítulo é explicar com mais detalhes o que é Conformidade Arquitetural de *Software* a fim de solidificar por que é essencial realizarmos a Análise de Conformidade durante o desenvolvimento de um sistema. Para isso, falamos mais sobre arquitetura de *software*, apresentando definições de arquitetura presentes na literatura e destrinchando um padrão de arquitetura bastante utilizado como exemplo.

Apresentamos também as metodologias e ferramentas existentes atualmente para auxiliar na Análise de Conformidade Arquitetural e critérios segundo os quais comparamos essas metodologias entre si e, posteriormente, com a metodologia que desenvolvemos neste trabalho.

2.1 Arquitetura de *Software*

Para entender do que se trata Conformidade Arquitetural de *Software*, precisamos compreender o que é arquitetura de *software*. Encontramos várias definições de arquitetura de *software* na literatura. Mencionamos algumas a seguir:

1. De acordo com (IEEE, 2007), arquitetura é a organização fundamental de um sistema, que é constituída por seus componentes, as relações entre esses componentes e com o ambiente, e os princípios que guiam seu *design* e evolução.
2. Ian Sommerville, em (SOMMERVILLE, 2011), explica que a arquitetura de *software* deve preocupar-se em explicar como um sistema está organizado e representar a estrutura geral desse sistema. A arquitetura também deve especificar os relacionamentos e as comunicações entre os componentes que constituem o sistema.
3. Perry e Wolf em (PERRY; WOLF, 1992) definem arquitetura como sendo a equação mostrada a seguir e a explicação dos seus termos:

   Arquitetura = {Elementos, Organização, Decisões}

   Segundo essa definição, a arquitetura de *software* é o conjunto de elementos arquiteturais que possuem alguma organização. Tais elementos e a maneira como devem ser organizados são definidos por decisões baseadas nos objetivos e nas restrições do *software*.

4. A definição proposta por (BASS *et al.*, 2003) diz que a arquitetura de um programa ou de sistemas computacionais é a estrutura ou estruturas do sistema, a qual é composta de elementos de software, as propriedades externamente visíveis desses elementos e os relacionamentos entre eles.

Na Subseção 1.2.4, apresentamos uma definição de arquitetura (Definição 1.2.3) baseada nas definições mostradas acima. Nessa definição, resolvemos deixar claro a importância da documentação. A documentação é fundamental para a qualidade do *software*. Para o problema com o qual estamos lidando neste texto, é essencial termos uma ótima documentação da arquitetura para facilitar o processo de formalização das especificações. Por isso, decidimos destacar esse ponto.

Definir a arquitetura traz uma série de benefícios ao processo de desenvolvimento de *software*:

1. Melhora a comunicação entre as partes interessadas no *software*;
2. Possibilita análise prévia, o que ajuda a prevenir erros;
3. Auxilia no reuso de *software*;
4. Melhora a qualidade da documentação do *software*.

Dessa forma, a arquitetura de *software* é fundamental para construir *softwares* de maneira eficiente, economizando tempo e dinheiro e garantindo a qualidade do sistema desenvolvido.

Na Subseção 1.2.4, apresentamos a arquitetura de repositório1.2.4. Explicamos agora outro padrão arquitetural e voltamos a falar sobre esse padrão quando mostrarmos exemplos de formalização de especificações na seção seguinte.

Exemplo 2 (Arquitetura de camadas) *Um exemplo de arquitetura bastante comum é a arquitetura de camadas (SOMMERVILLE, 2011). Esse padrão organiza o sistema em camadas separadas e cada camada depende dos serviços e recursos oferecidos pela camada imediatamente abaixo dela. Desde que as interfaces de comunicação não sejam alteradas, podemos mudar a implementação de uma camada ou adicionar novas funcionalidades sem precisar alterar as outras camadas. De fato, podemos substituir uma camada inteira por outra equivalente*

sem a necessidade de alterar qualquer outra parte do sistema. Isso torna o sistema fácil de manter.

Uma das vantagens dessa arquitetura é que ela facilita desenvolver *softwares* para diferentes máquinas ou sistemas operacionais. Nas camadas mais internas, implementamos as funcionalidades que são especificas para um determinado tipo de máquina ou para um sistema operacional específico. As aplicações do *software* em si ficam nas camadas mais externas. Logo, podemos facilmente reaproveitá-las ao desenvolver uma versão do *software* para diversas máquinas e sistemas operacionais. Essa característica dessa arquitetura permite a construção de sistemas altamente portáveis.

Nesta arquitetura, almejamos que uma camada comunique-se apenas com as suas camadas vizinhas. Uma camada deve usar os recursos da camada que estiver abaixo e prover recursos para a camada que estiver acima. Se por acaso uma camada precisar comunicar-se com uma camada que não é vizinha, a mensagem deve ser processada e repassada pelas camadas intermediárias, evitando a comunicação direta.

A tabela a seguir, retirada de (SOMMERVILLE, 2011), sintetiza o padrão de camadas:

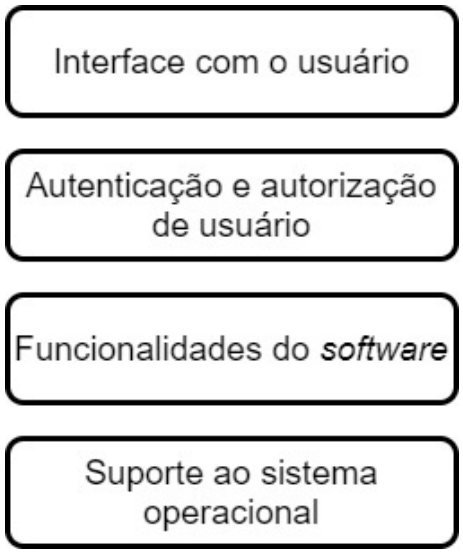
A figura a seguir representa um modelo genérico de um *software* com arquitetura de camadas. Temos quatro camadas:

1. A camada mais abaixo realiza a comunicação com o sistema operacional.
2. A camada acima dessa implementa as funcionalidades do *software*, ou seja, ela é responsável pela implementação das operações que sistema deve fazer.
3. Temos uma camada encarregada pelas operações de segurança, garantindo que apenas usuários autorizados consigam acessar o sistema.
4. A camada mais externa possui a interface com o usuário, ou seja, a função dessa camada é realizar a comunicação entre o usuário e o sistema.

O exemplo a seguir mostra a aplicação da arquitetura de camadas ao sistema de uma biblioteca. Esse sistema controla o acesso digital a livros e afins com direitos autorais de um grupo de bibliotecas de uma universidade. Temos cinco camadas, sendo que a camada mais abaixo é responsável pela comunicação com o banco de dados de cada biblioteca. A camada

Padrão de arquitetura de camadas	
Descrição	Organiza o sistema em camadas com determinadas funcionalidades associadas a cada camada. Uma camada provê serviços para a camada acima dela. As camadas mais baixas são responsáveis pela comunicação com o sistema operacional, bancos de dados.
Exemplo	Apresentamos um exemplo na imagem **??**.
Quando usar	Usado quando adicionamos novas funcionalidades a um sistema já existente, quando o desenvolvimento está espalhado entre vários times com cada time sendo responsável por uma camada, quando um mecanismo de segurança de vários níveis é necessário.
Vantagens	Permite a substituição de camadas inteiras desde que as interfaces de comunicação sejam mantidas, funcionalidades redundantes, como por exemplo autenticação, podem ser inseridas em cada camada para aumentar a confiabilidade e a segurança do sistema.
Desvantagens	Definir uma separação clara entre as camadas é difícil, uma camada de nível mais alto pode precisar se comunicar com uma camada de nível mais baixo do que a camada imediatamente abaixo, desempenho pode ser um problema se uma requisição precisar ser processada por várias camadas.

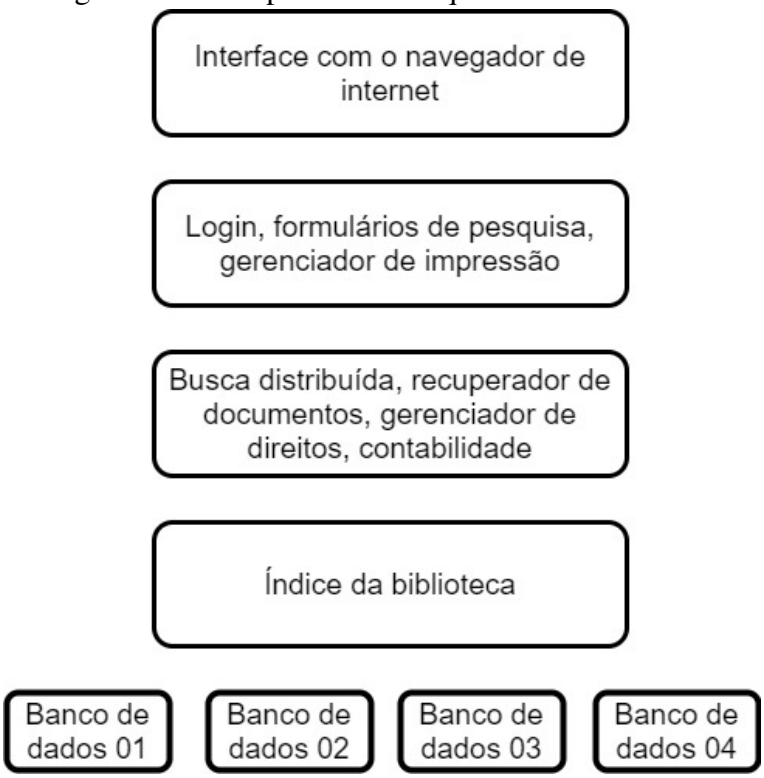
Figura 5 – Modelo genérico de arquitetura de camadas



Fonte: elaborado pelo autor (2021).

mais externa realiza a comunicação com o usuário através de um navegador. Este exemplo foi retirado de (SOMMERVILLE, 2011).

Figura 6 – Exemplo de uma arquitetura em camadas



Fonte: elaborado pelo autor (2021).

Agora que definimos o que é arquitetura de *software*, explicitamos a sua importância e mostramos alguns exemplos, na seção a seguir, apresentamos um problema que surge durante a implementação da arquitetura a fim de elucidar melhor o objetivo deste trabalho.

2.2 Análise de Conformidade Arquitetural

Existem duas perspectivas para a arquitetura de *software* (PERRY; WOLF, 1992):

1. Perspectiva de *design*: é a arquitetura definida no projeto do software, chamada de arquitetura prescritiva.
2. Perspectiva de implementação: é a arquitetura que de fato está sendo implementada à medida que os desenvolvedores escrevem os códigos-fontes do *software*. É chamada de arquitetura descritiva.

Idealmente, a arquitetura descritiva corresponde exatamente à arquitetura prescritiva. Ou seja, a arquitetura que os desenvolvedores estão implementando corresponde precisamente à arquitetura determinada no projeto do sistema. Entretanto, no decurso de desenvolvimento,

surgem discrepâncias entre essas perspectivas, o que faz com que a implementação divirja do projeto. Esse problema é chamado de degradação arquitetural (SILVA; BALASUBRAMANIAM, 2012; PASSOS *et al.*, 2010) e ocasiona problemas ao *software* ao introduzir dependências não intencionais entre os componentes do *software* e ao violar os princípios de modularidade, que são importantes para o reuso de *software*, por exemplo.

Reusar componentes de *software* permite reduzir o tempo e os custos de desenvolvimento (SOMMERVILLE, 2011). Logo, é sempre desejável construir componentes que sejam reutilizáveis e reusar componentes já implementados. Por isso, garantir que o *software* que está sendo produzido segue as diretrizes de modularidade definidas na fase de projeto é de extrema importância.

Outros problemas ocasionados pela degradação arquitetural incluem:

1. Perda da escalabilidade planejada;
2. Redução da manutenibilidade, o que torna difícil fazer atualizações e correções;
3. Diminuição da confiabilidade.

Para lidar com a degradação arquitetural, temos a Análise de Conformidade Arquitetural (SILVA; BALASUBRAMANIAM, 2012; FILHO, 2016; PASSOS *et al.*, 2010). Essa análise consiste em mensurar o quanto um software está alinhado com aquilo que foi planejado. Para verificar a conformidade arquitetural, precisamos analisar a arquitetura descritiva (implementação) contra as decisões de projeto contidas na arquitetura prescritiva (*design*). A conformidade arquitetural é alcançada quando não existe divergência entre a arquitetura real e a arquitetura planejada.

Durante o processo de desenvolvimento, a conformidade entre as arquiteturas é periodicamente examinada. Esse é um processo complexo, pois envolve analisar decisões de *design* e artefatos de *software* em alguns níveis:

1. Nível mais alto de abstração:
   - Módulos;
   - Conectores;
   - Interfaces.
2. Nível de detalhamento mais profundo de implementação de software:
   - Métodos;

- Classes;
- Pacotes;
- Estruturas de fluxo de controle.

Ferramentas para auxiliar nesse processo são fundamentais: reduzem o tempo de verificação, gastos, e são mais confiáveis do que a análise totalmente humana, inviável para sistemas grandes. Sendo assim, vamos aplicar Verificação de Modelos no nível de detalhamento mais profundo, na análise de propriedades relacionadas aos métodos, às classes e aos pacotes. Pacotes são conjuntos de classes relacionadas. Logo, verificar propriedades referentes a classes envolve verificar propriedades de pacotes. Para fazer isso, usamos os grafos de chamadas (Subseção 1.2.3). Acreditamos que essa metodologia funcione como uma ferramenta útil para a Análise de Conformidade Arquitetural.

Exemplo 3 (Especificações sobre a arquitetura de camadas) *Na seção anterior, falamos sobre a arquitetura de camadas 2.1. Mostramos neste exemplo algumas especificações que devem ser verificadas a respeito desse padrão arquitetural durante a Análise de Conformidade Arquitetural de uma implementação dele.*

Conforme falamos na explicação da arquitetura de camadas, precisamos garantir que as comunicações entre as camadas ocorram somente com as suas vizinhas. Considere uma classe A de uma camada i e uma classe B de uma camada i + j, onde j ≥ 2. Em outras palavras, B não é vizinha de A. A seguinte especificação não deve ser satisfeita pela sistema:

A classe A **chama diretamente a classe** B: isso significa que encontramos no grafo de chamadas um método da classe A realizando uma chamada a um método da classe B.

Na arquitetura de camadas, requisições podem passar por várias camadas. Ao receber uma requisição, uma camada deve processar se ela pode atender a essa requisição ou deve repassá-la para a camada inferior. Esse ponto também é importante para avaliar o desempenho da troca de mensagens do sistema, o que pode se tornar um problema. Neste contexto, pode ser útil verificar a seguinte especificação:

A classe A **chama indiretamente a classe** B: isso significa que encontramos no grafo

de chamadas um método da classe A que chama um método de uma classe C, onde $C \neq A$ e $C \neq B$ e após uma sequência de chamadas, temos uma chamada para um método da classe B.

Seja m um método de A e n, um método de B. O método m pode ser responsável por repassar uma requisição para a camada adequada. Assumindo que essa requisição é tratada pelo método n da classe B, a seguinte especificação deve ser satisfeita pelo sistema:

Existe uma sequência de chamadas partindo do método m que alcança o método n: isso significa que m chama um método qualquer, esse método chama outro método, que pode ser n ou não, e assim sucessivamente até encontrarmos n.

Podemos ainda verificar questões relacionadas a ciclos de chamadas usando as especificações a seguir:

- **A classe A está em um *loop* de chamadas**: isso significa que temos um método da classe A que chama algum método de uma classe qualquer. Esse método pode chamar ou não um método da classe A, o que importa é que uma chamada a um método da classe A sempre volte a acontecer durante a execução. A Figura 25 ilustra essa situação.
- **Existe uma sequência de chamadas partindo do método m que alcança o método m**: isso significa que m chama um método qualquer, esse método chama outro método, que pode ser m ou não, e assim sucessivamente até retornamos para m.

Todas essas especificações são avaliadas com base do grafo de chamadas do *software* e mostramos como formalizá-las usando lógica no Capítulo 4.

Na seção a seguir, apresentamos as principais técnicas e abordagens para realizar Análise de Conformidade Arquitetural.

2.3 Estado da Arte da Análise de Conformidade Arquitetural

O objetivo dessa seção é apresentar as principais metodologias para realizar Análise de Conformidade Arquitetural encontradas na literatura. Segundo (SCHRÖDER, 2020), temos cinco abordagens gerais que contemplam essas metodologias:

1. Abordagens baseadas em modelos de reflexão;
2. Abordagens baseadas em regras;
3. Abordagens baseadas em lógica;
4. Abordagens baseadas em consultas;
5. Especificações embutidas.

Falamos sobre cada uma dessas abordagens nesta seção e elencamos técnicas, métodos e ferramentas que se enquadram em cada abordagem. No final, explicamos alguns critérios de avaliação e os usamos para comparar as diversas metodologias e ferramentas existentes.

2.3.1 Abordagens baseadas em modelos de reflexão

Modelos de reflexão (MURPHY *et al.*, 1995) especificam a arquitetura como um modelo de alto nível. Esse modelo possui os componentes arquiteturais e as relações entre eles. Esse modelo é comparado com o modelo da arquitetura que de fato está sendo implementada. Isso é feito por meio de um grafo de dependências do *software*: os vértices representam componentes e as arestas, as relações entre esses componentes. (Grafos de chamadas 1.2.3 são um tipo de grafos de dependências).

Temos dois modelos: um modelo de alto nível da arquitetura planejada e o modelo da implementação. Para compará-los, é necessário mapear o modelo de implementação para o modelo de alto nível. Após isso, as diferenças entre os modelos são calculadas e representadas como um modelo de reflexão, no qual as arestas encaixam-se nas seguintes categorias:

Convergência: um componente ou uma relação está implementado de acordo com a arquitetura planejada.

Divergência: um componente ou uma relação viola o que está definido na arquitetura planejada.

Ausência: um componente ou uma relação que está ausente na implementação, ou seja, algo que foi planejado, porém não foi implementado.

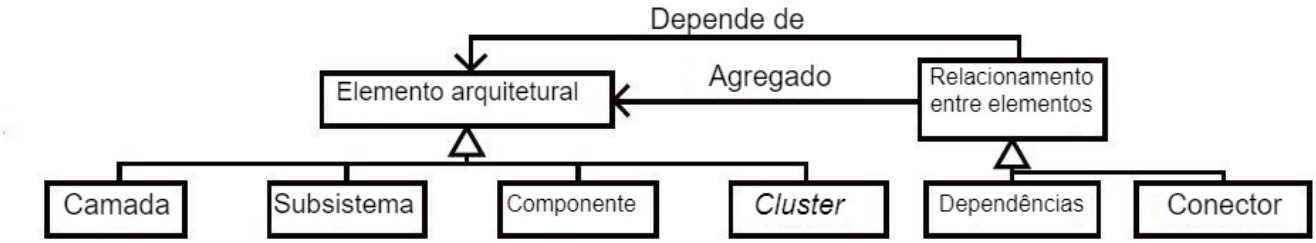
As seguintes ferramentas seguem a abordagem de modelos de reflexão:

1. Sonorgraph (SONARGRAPH-ARCHITECT, 2017);
2. Software Architecture Visualization and Evaluation (SAVE) (DUSZYNSKI *et al.*, 2009);

3. Structure101 (STRUCTURE. . . ,);
4. Teamscale (DEISSENBOECK *et al.*, 2010).

As ferramentas mencionadas acima possuem suporte para a modelagem da arquitetura. Por meio de uma notação gráfica, as ferramentas mencionadas facilitam a compreensão da arquitetura e das especificações. Como exemplo, considere a imagem a seguir, que representa os elementos disponíveis na ferramenta SAVE para modelar a arquitetura:

Figura 7 – Elementos da ferramenta SAVE



Fonte: elaborado pelo autor (2021).

A ferramenta SAVE modela a arquitetura usando camadas, subsistemas, componentes e *clusters*. Todos esses são considerados elementos arquiteturais. Para descrever os relacionamentos entre eles, são usados conectores e dependências. Essa ferramenta dá suporte para as linguagens de programação Java, C++ e Delphi.

Segundo (SCHRÖDER, 2020), o problema das quatro ferramentas mencionadas é que a linguagem usada para definir a arquitetura é fixa: se for necessário adicionar novos elementos para descrever a arquitetura pretendida, é difícil modificar a linguagem.

2.3.2 Abordagens baseadas em regras

Abordagens baseadas em regras descrevem a arquitetura por meio de regras arquiteturais. Para especificar essas regras, são usadas linguagens específicas de domínio (DSL, da sigla em inglês). As DSL permitem que os arquitetos de *software* escrevam as restrições e as dependências existentes no sistema.
As ferramentas listadas a seguir seguem a abordagem baseada em regras:

1. DCL (TERRA; VALENTE, 2009);

2. Macker (MACKER,);
3. StyleBasedChecker (BECKER-PECHAU, 2014);
4. Dictō (CARACCIOLO *et al.*, 2015);
5. Lattix Architect (LATTIX. . . ,);
6. HUSACCT (PRUIJT *et al.*, 2014);
7. ArCatch (FILHO, 2016);

Cada uma dessas ferramentas possui a sua própria DSL para descrever a arquitetura de *software*, com semântica e sintaxe próprias. Como exemplo, considere a DCL. A linguagem específica de domínio dessa ferramenta permite escrever as restrições entre os componentes de arquitetura. Temos como expressar proibições, permissões, obrigações e palavras-chave para especificar elementos arquiteturais e elementos a nível do código-fonte. Como exemplo, considere como expressamos uma regra de dependência em um sistema de arquitetura de camadas:

Figura 8 – Regra de dependência

```
module Model: org.company.model.**
module View: org.company.view.**
Model cannot-depend View
```

Fonte: elaborado pelo autor (2021).

Temos duas camadas: Model e View. Isso é especificado utilizando a palavra-chave module. View é uma camada superior à camada Model. Logo, Model não pode depender de View, conforme explicamos quando falamos sobre arquitetura de camadas 2.1. Por isso, temos a regra cannot-depend.

Temos um método focado no tratamento de exceção, i.e, ocorrências de condições excepcionais durante a execução do *software* (DEITEL; DEITEL, 2014), em (FILHO, 2016). A ferramenta ArCatch visa combater a degradação arquitetural que acontece no âmbito de tratamento de exceção, provendo uma maneira de documentar decisões de *design* e a utilização dessas na análise de conformidade do código-fonte.

De acordo com (SCHRÖDER, 2020), as abordagens baseadas em regras fornecem uma sintaxe clara das suas DSL, o que resulta em formalizações fáceis de compreender. Entretanto, modificar a linguagem para adicionar novos elementos é complicado. Em geral, as ferramentas dessa abordagem não possuem suporte para modelar a arquitetura, como elementos gráficos. Toda a especificação arquitetural é feita usando regras.

2.3.3 Abordagens baseadas em lógica

Abordagens baseadas em lógica usam o formalismo das lógicas para especificar regras de arquitetura. Segundo (MENS; MENS, 2000), regras de arquitetura podem ser mapeadas facilmente para predicados. As metodologias a seguir se enquadram nesse tipo de abordagem:

1. LogEn (MITSCHKE *et al.*, 2013);
2. *Structural Constraint Language* (SCL) (HOU; HOOVER, 2006);
3. Herold (HEROLD, 2011);
4. Call Graph and Model Checking for Fine-Grained Android Malicious Behaviour Detection (IADAROLA *et al.*, 2020);

Herold (HEROLD, 2011) propõe uma metodologia com base na lógica de primeira ordem (RYAN; HUTH, 2004; EBBINGHAUS *et al.*, 1994). Lógica de primeira ordem possui flexibilidade na definição de regras arquiteturais, algo que, falta em outras abordagens, conforme apontamos nas subseções anteriores. O conjunto de predicados que representam abstrações arquiteturais não é predefinido: predicados podem ser adicionados conforme a necessidade dos engenheiros de *software*. Regras arquiteturais são descritas por meio de fórmulas de primeira ordem, codificando as propriedades que o sistema sendo avaliado deve possuir.

A metodologia SCL (HOU; HOOVER, 2006) define fórmulas de primeira ordem que podem ser avaliadas com base no código-fonte. Com isso, é possível avaliar as restrições impostas a programas escritos em C++ e Java. Predicados são definidos com base no código-fonte e não no projeto da arquitetura, como acontece com a metodologia proposta do Herold. Os predicados são predefinidos.

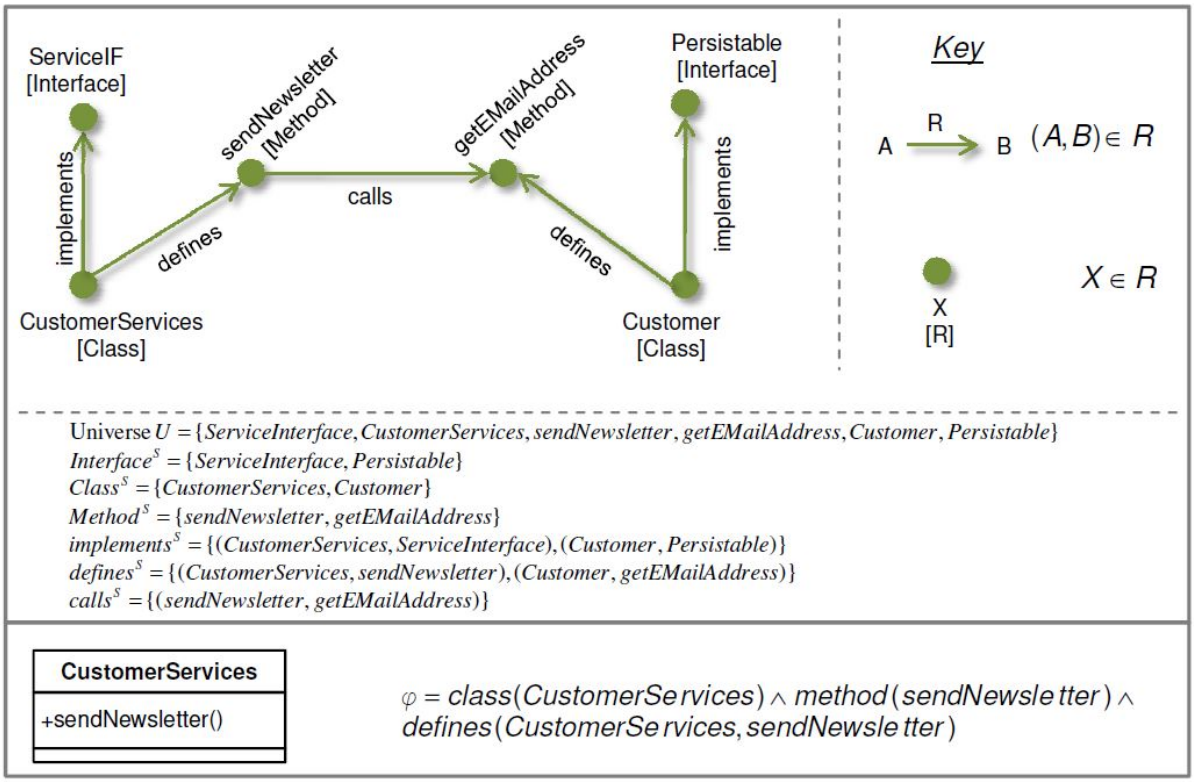
LogEn possui os chamados *ensemblies*. *Ensemblies* permitem a combinação entre código-fonte e o projeto de alto nível da arquitetura, o que leva a uma visão mais refinada do código-fonte. A linguagem da ferramenta permite definir dependências entre os *ensemblies* para representar as dependências existentes no sistema. O código-fonte é verificado de acordo com essas especificações.

Em (IADAROLA *et al.*, 2020), um método baseado em Verificação de Modelos para detectar *malwares* no sistema operacional de dispositivos móveis Android é proposto. Uma das estruturas usadas para representar as aplicações Android é o grafo de chamadas. As propriedades são formalizadas usando lógica μ-calculus.

Em (QUEIROZ *et al.*, 2014), temos uma metodologia que avalia várias versões de uma ontologia usando uma lógica temporal hibridizada. O espaço de versões de ontologia é representado por um grafo direcionado. Por meio de operadores da lógica híbrida, é possível comparar duas versões para saber se elas concordam ou discordam em alguma propriedade. O método proposto no artigo percorre o grafo de ontologias disparando consultas nos vértices. Entretanto, não existe a divisão em dois níveis que se comunicam entre si que propomos no presente trabalho. O trabalho desenvolvido em (QUEIROZ *et al.*, 2014) não foca em Conformidade Arquitetural: trata-se de um trabalho focado em ontologias visando aplicações em Web Semântica e não para Análise de Conformidade Arquitetural. Para aplicá-lo na Conformidade Arquitetural, seria necessário representar o problema de conformidade como ontologias. Por causa disso, não o incluímos na tabela comparativa do final desta seção. Consideramos importante mencioná-lo aqui, já que o trabalho também usa uma combinação de lógica temporal com uma lógica híbrida.

Como exemplo, mostramos como a metodologia proposta por Herold (HEROLD, 2011) modela um sistema usando lógica de primeira ordem. Temos duas classes, CustomerServices e Customer, dois métodos e duas interfaces. Cada classe deve implementar uma interface e definir um método. Usamos palavras reservadas para indicar isso, a saber, *implements* e *defines*. Também existe uma chamada entre os métodos, o que é representado pela palavra reservada *calls*. Isso é modelado por meio de uma estrutura de primeira ordem, com o conjunto universo sendo as classes, os métodos e as interfaces. Para indicar o que é classe, método e interface, bem como as relações definidas pelas palavras reservadas, usamos relações. O método sendNewsletter deve ser implementado pela classe CustomerServices. A fórmula apresentada na figura a seguir formaliza isso: CustomerServices deve ser uma classe, sendNewsletter um método e essa classe deve definir esse método.

Figura 9 – Exemplo de uso da metodologia de Herold



Fonte: elaborado pelo autor (2021).

Conforme elencado por (SCHRÖDER, 2020), abordagens baseadas em lógica são facilmente estendidas e muito expressivas no quesito descrição e formalização de arquitetura. Novos conceitos arquiteturais são adicionados pela inclusão de novos predicados. Contudo, essas abordagens são complicadas de usar e difíceis de entender: engenheiros de *software* precisam ter expertise em lógica matemática para definir os predicados e especificar as regras.

2.3.4 Abordagens baseadas em consultas

Abordagens baseadas em consultas fazem consultas no código-fonte com o objetivo de obter elementos e estruturas. Com isso, é possível analisar estilos de códigos, procurar erros e implementações que não estão de acordo com as especificações e regras arquiteturais. Uma regra de arquitetura é violada quando o resultado de uma consulta não é vazio. As seguintes metodologias se baseiam em consultas:

1. *Code Query Language* (CQLinq) (CQLINQ. . . ,);
2. .QL (MOOR *et al.*, 2007);
3. jQassistant (JQASSISTANT,);

A CQLinq utiliza uma linguagem semelhante à linguagem SQL: o padrão SELECT FROM WHERE da SQL é usado pela CQLinq para formular consultas. Essa metodologia permite a execução de consultas em códigos-fonte de linguagens orientadas a objetos. As consultas são flexíveis e fáceis de serem escritas. Contudo, conceitos arquiteturais de alto nível não são suportados. A .QL é semelhante à CQLinq. Essa ferramenta permite a definição de predicados para serem utilizados nas consultas e a sintaxe da sua linguagem é semelhante a Java, o que contribui para um rápido aprendizado do uso da ferramenta.

A jQassistant baseia-se em uma ferramenta que utiliza um banco de dados de grafos e as consultas são realizadas nesses grafos. O código-fonte é armazenado como um grafo no banco de dados e consultas são executadas a fim de encontrar violações às especificações arquiteturais. Elementos do código-fonte, como classes ou *headers*, são representados como vértices e as dependências entre eles, como arestas direcionadas.

Como exemplo, considere a arquitetura de camadas 2.1. Mostramos como representar a regra de dependência entre as camadas que foi apresentada na Figura 8 utilizando a jQassistant. A regra diz que a camada Model não pode depender da camada View. Primeiramente, é necessário definir as camadas Model e View. A imagem a seguir mostra como fazer isso para a camada Model:

Figura 10 – Exemplo de consulta com a jQassistant

```
<concept id="Model">
  <cypher>
    MATCH
      model:Package
    WHERE
      model.name = "org.company.model"
    SET
      model:ModelLayer
  </cypher>
</concept>
```

Fonte: elaborado pelo autor (2021).

Em seguida, precisamos definir quais dependências são permitidas entre as camadas. Suponha que existe uma camada chamada Logic logo abaixo da camada View. Então, conforme o padrão de arquitetura de camadas, existe uma dependência entre essas duas camadas. Essa definição é exibida a seguir:

Figura 11 – Exemplo de consulta com a jQassistant

```
<concept id="DefinedDependencyViewLogic">
  <cypher>
    MATCH
      view:Layer
    MATCH
      logic:Layer
    CREATE UNIQUE
      (view)-[:defines-dependency]->(logic)
  </cypher>
</concept>
```

Fonte: elaborado pelo autor (2021).

A regra arquitetural é definida pela consulta ilustrada a seguir, escrita na linguagem da ferramenta jQassistant. Essa consulta retorna todas as camadas para as quais são encontradas relações de dependência. Logo, se Model e View estiverem na lista de camadas retornadas pela consulta, a especificação do modelo de arquitetura de camadas é violada.

Figura 12 – Exemplo de consulta com a jQassistant

```
<constraint id="UndefinedDependency">
  <cypher>
    MATCH
      (layer1:Layer) -[:depends-on]->(layer2:Layer)
    WHERE NOT
      (layer1)-[defines-dependency]->(layer2) AND
      layer1.name <> layer2.name
    RETURN
      layer1.name, layer2.name
  </cypher>
</constraint>
```

Fonte: elaborado pelo autor (2021).

De acordo com (SCHRÖDER, 2020), as abordagens baseadas em consultas são uma maneira poderosa e flexível para definir e verificar regras de arquitetura. A flexibilidade dessas abordagens permite verificar diversos aspectos das arquiteturas de sistemas. Entretanto, essas abordagens não conseguem descrever modelos arquiteturais, conforme fazemos visualmente nas figuras 4 e 5, por exemplo. Outra desvantagem dessa abordagem, é a sua verbosidade: compare o exemplo apresentado aqui com o exemplo mostrado nas abordagens baseadas em regras. Temos que definir muitos conceitos e a sintaxe da linguagem é verbosa. Consequentemente, abordagens

baseadas em consultas são mais complicadas de entender.

2.3.5 Especificações embutidas

Ferramentas que utilizam especificações embutidas adicionam as regras arquiteturais diretamente no código-fonte, como comentários ou anotações. O objetivo desse tipo de abordagem é reduzir o espaço entre o projeto de arquitetura e a implementação, colocando as especificações arquiteturais o mais próximo possível do código-fonte. Isso facilita a análise do impacto de mudanças na implementação a fim de saber se as mudanças afetam as regras arquiteturais.

Como exemplo de ferramentas que usam especificações embutidas, temos:

1. ArchFace (UBAYASHI *et al.*, 2010);
2. ArchJava (ALDRICH *et al.*, 2002).

Segundo (SCHRÖDER, 2020), especificações embutidas possuem uma desvantagem crucial: é muito trabalhoso estender a linguagem com novos conceitos arquiteturais. Por exemplo, fazer isso na ArchJava envolve modificar a sintaxe da Java, o que acarreta em modificações nos ambientes de programação e compiladores. Isso tende exigir muito esforço e necessita de especialistas em Java.

2.3.6 Critérios de avaliação

Elencamos a seguir os critérios usados para comparar as diferentes metodologias para realizar Análise de Conformidade Arquitetural existentes. Os critérios de 1 a 4 são os propostos por (SCHRÖDER, 2020). Os critérios 5 e 6 são sugestões deste trabalho.

1. Flexibilidade da linguagem de modelagem: a linguagem de modelagem usada pela metodologia deve ser flexível para capturar os conceitos e as definições envolvidas no projeto de arquitetura. O ideal é que a linguagem tenha mecanismos de extensão que permitam a inclusão de novos elementos arquiteturais conforme a necessidade dos engenheiros de *software*. Para ser flexível, a linguagem deve permitir que o usuário a personalize de acordo com a sua necessidade: os nomes usados para se referir aos elementos arquiteturais devem ser arbitrariamente escolhidos pelo usuário e não predefinidos na linguagem e a semântica dos conceitos e definições da arquitetura não devem ser ambíguos.

2. Suporte à modelagem arquitetural: modelos arquiteturais são importantes para o entendi-

mento e para a documentação da arquitetura. Eles também são fundamentais para que melhorias do sistema possam ser realizadas. A metodologia deve prover alguma maneira de modelar e descrever a arquitetura de *software* pretendida.

3. **Suporte para integração de documentação de arquitetura:** a descrição da arquitetura pretendida não deve estar atrelada à ferramenta de Análise de Conformidade Arquitetural utilizada. Dessa forma, a arquitetura é acessível para qualquer pessoa. Idealmente, a ferramenta possui alguma maneira de acessar a arquitetura planejada e mantê-la próxima ao código-fonte, que corresponde à implementação. Uma maneira de fazer isso é utilizar sistemas de controle de versões.
4. **Compreensibilidade:** a metodologia deve utilizar linguagens que sejam fáceis de ler e entender. Formalizar as especificações deve ser fácil e simples. Linguagens verbosas ou complicadas tornam a compreensão, e consequentemente o uso da metodologia, por parte dos engenheiros de *software* uma tarefa trabalhosa, o que pode afastá-los da metodologia em questão, além aumentar as chances de cometer erros.
5. **Suporte à verificação em dois níveis:** conforme explicamos na Seção 1.3, este trabalho lida com dois níveis de verificação, um referente a uma versão de *software* em particular e um nível global, que analisa todas as versões. Este critério analisa se a metodologia trata desses dois níveis ou apenas do nível referente a uma única versão.
6. **Manutenção do histórico das versões:** analisamos se a metodologia propõe uma maneira de manter o histórico das versões de *software* e das análises de conformidade executadas em cada versão. Se a ferramenta não disponibilizar um mecanismo para isso, podemos sempre usar um Sistema de Controle de Versões e gerenciar esse processo por conta própria, sem o suporte da ferramenta de Análise de Conformidade utilizada.

As metodologias de Análise de Conformidade Arquitetural precisam ser flexíveis para representarem os conceitos que são introduzidos durante o desenvolvimento de *software*. Esses conceitos mudam constantemente, à medida que novas tecnologias são desenvolvidas, por exemplo. Logo, é fundamental que a linguagem de modelagem seja flexível para acoplar mudanças. Por isso, a atenção dada ao critério 1. Se a linguagem para a Análise de Conformidade Arquitetural também permitir a modelagem arquitetural, temos uma linguagem única para os dois propósitos, o que facilita o processo de aprendizagem e agiliza o processo de modelagem e verificação. Por isso, consideramos o critério 2. Caso não seja possível modelar a arquitetura

com a ferramenta de Análise de Conformidade, é preciso existir alguma maneira de integrar a arquitetura modelada com a ferramenta para que a verificação seja realizada. Por isso, temos o critério 3. A metodologia deve ser simples de entender, seja para compreender projetos já existentes, seja para aplicá-la em novos projetos. Esse ponto também influencia diretamente na curva de aprendizado da metodologia, permitindo economizar tempo no processo de desenvolvimento. Logo, analisamos o critério 4.

Os critérios 5 e 6 são parte da inovação que propomos neste trabalho. As ferramentas e metodologias que encontramos na literatura visam realizar a verificação de especificações referentes a apenas uma versão do *software*, sem contemplar uma visão global do processo e sem levar em consideração o histórico de desenvolvimento e especificações a respeito desse histórico. A verificação em nível global, contemplando o histórico de versões, é uma contribuição deste trabalho. Logo, o critério 5 avalia se a metodologia realiza verificações apenas em uma versão específica ou se possuiu suporte para realizar a verificação no histórico de versões. Para isso, é importante que a metodologia tenha uma maneira de manter o histórico de versões. A metodologia proposta neste trabalho utiliza o grafo de versões para essa finalidade. O critério 6 avalia esse ponto.

Apresentamos a seguir uma tabela para comparar as metodologias e ferramentas para análise de conformidade Arquitetural mencionadas nesta seção usando os critérios elencados acima. Essa tabela deve-se a (SCHRÖDER, 2020). Voltamos a essa tabela depois de apresentar a metodologia proposta neste trabalho para compará-la com as metodologias que já existem. Para cada critério elencado, consideramos três possibilidades, conforme (SCHRÖDER, 2020):

+: requisitos do critério atendidos;
o: requisitos do critério parcialmente atendidos;
-: requisitos do critério não atendidos.

Sumarizamos a comparação entre as metodologias na tabela da página a seguir. Não encontramos na literatura qualquer metodologia que se preocupa em executar a Análise de Conformidade Arquitetural levando em consideração todas as versões do *software*: a preocupação é sempre com apenas uma versão em particular. As metodologias também não proveem um mecanismo para armazenar o histórico das versões. Entretanto, conforme mencionamos

anteriormente, podemos usar os Sistemas de Controle de Versões: as ferramentas não apresentam impedimentos quanto a isso. Por isso, consideramos que elas atendem parcialmente ao critério 6. Atente que esse critério não é sobre a verificação de especificações e sim se a metodologia leva em consideração alguma estrutura para manter o histórico das versões de um *software*. O critério sobre verificação de propriedades é o critério 5, que analisa se a metodologia é voltada para verificação apenas em uma versão em particular ou se a metodologia também leva em consideração o processo global de desenvolvimento de *software*.

Critério	1	2	3	4	5	6
Modelos de reflexão						
Sonorgraph	-	+	-	+	-	o
SAVE	-	+	-	+	-	o
Structure101	-	+	-	+	-	o
Teamscale	-	+	-	+	-	o
Baseadas em regras						
DCL	-	-	+	+	-	o
TamDera	o	-	+	o	-	o
Dicto	+	-	+	+	-	o
Macker	o	-	+	o	-	o
Lattix Architect	-	+	-	+	-	o
StyleBasedChecker	+	-	+	o	-	o
ArCatch	+	-	+	o	-	o
Baseadas em lógica						
Herold	+	+	+	-	-	o
SCL	-	-	+	-	-	o
LogEn	o	-	+	-	-	o
Malicious Detection	-	-	+	o	-	o
Baseadas em consultas						
CQL	+	-	+	o	-	o
.QL	+	-	+	o	-	o
jQAssistant	+	-	+	o	-	o
Especificações embutidas						
ArchFace	-	-	-	o	-	o
ArchJava	-	-	-	o	-	o

3 VERIFICAÇÃO DE MODELOS

Este capítulo visa explicar com mais detalhes o que é o método da Verificação de Modelos. Apresentamos mais definições desse método, suas fases, esquematizamos o seu funcionamento e elencamos as vantagens e as desvantagens do método. Conforme explicamos no Capítulo 1, Verificação de Modelos requer uma maneira de modelar o sistema e uma linguagem formal para formalizar as propriedades que devem ser verificadas pelo método. Apresentamos as estruturas de Kripke, usadas na modelagem, e a lógica temporal CLT, usada para formalizar as propriedades.

Na metodologia que propomos neste trabalho, explicada no Capítulo 4, usamos também lógicas híbridas. Essas lógicas também podem ser usadas para formalizar as propriedades na Verificação de Modelos. Por isso apresentamos uma lógica híbrida neste capítulo.

Explicamos um algoritmo de Verificação de Modelos para a lógica CTL e que esse algoritmo pode ser utilizado também para a lógica híbrida apresentada.

3.1 Explicando o que é Verificação de Modelos

Neste texto, utilizamos o método formal Verificação de Modelos (*Model Checking*) para verificar propriedades e especificações de *software*. Segundo a referência (BAIER; KATOEN, 2008), Verificação de Modelos é uma técnica de verificação formal que permite confirmar se um dado sistema satisfaz determinadas propriedades por meio de uma inspeção sistemática dos estados do modelo que representa o sistema.

Nas últimas três décadas, métodos formais baseados em Verificação de Modelos para verificação da corretude de sistemas tornaram-se temas de pesquisas e possuem aplicações na indústria. Vamos apresentar as informações básicas a respeito desse método, dando um esboço sobre o seu funcionamento e aplicação na verificação de propriedades.

Verificação de Modelos é uma técnica eficiente para procurar erros no *design* de um sistema. Funciona particularmente bem para a verificação de propriedades relacionadas ao controle da execução de um programa. Dado um modelo M e uma propriedade devidamente formalizada α, perguntamos se a partir de um estado inicial s, M satisfaz α. Denotamos isso por $M, s \models \alpha$.

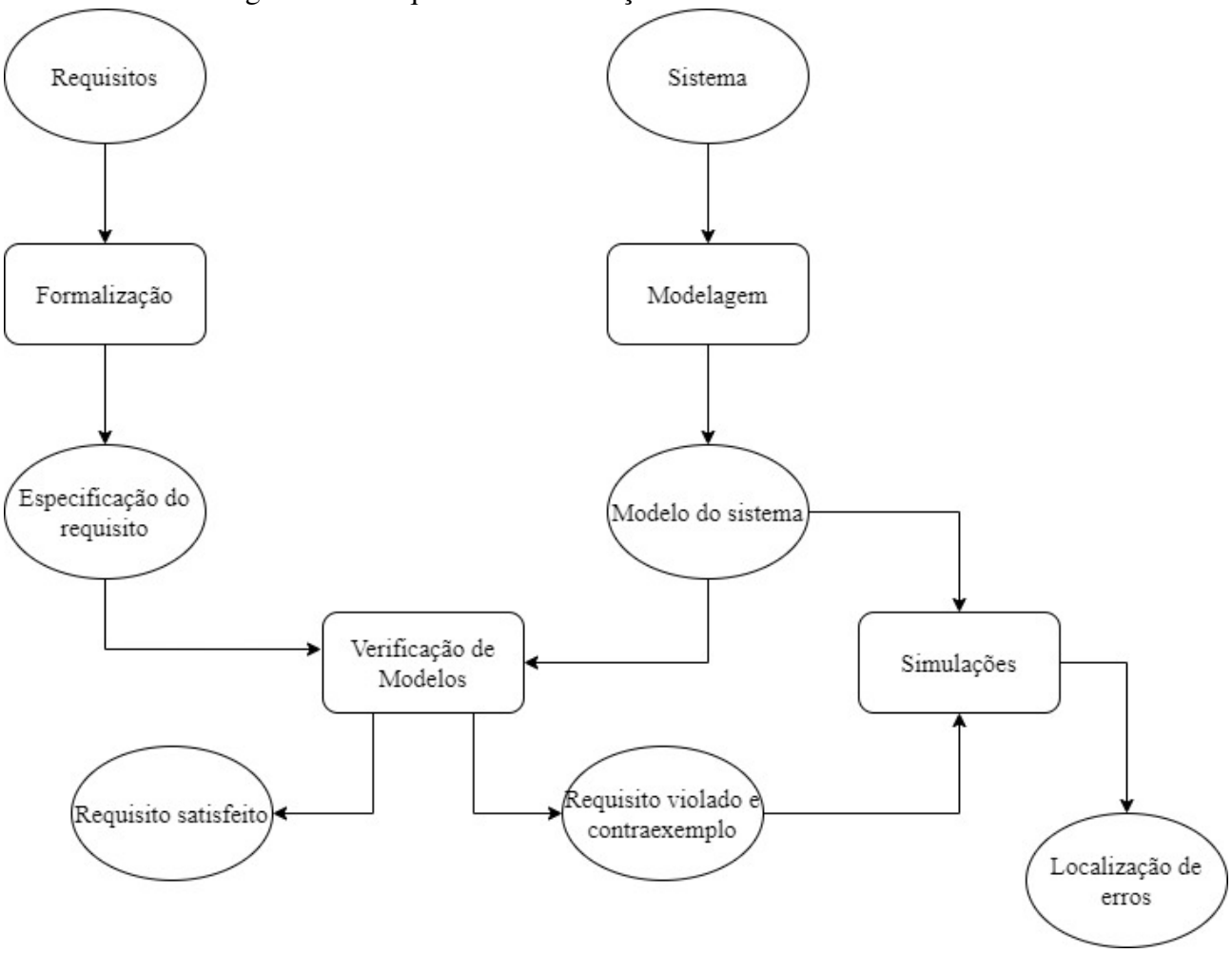
O processo de verificação de sistemas por Verificação de Modelos possui três fases:

1. Fase de modelagem: modelamos o sistema com o qual estamos trabalhando. Para isso, utilizamos alguma linguagem de descrição. Formalizamos as propriedades que desejamos verificar. Podemos usar uma lógica para essa finalidade.
2. Fase de execução: executamos um algoritmo de Verificação de Modelos para verificar se o modelo satisfaz às propriedades com as quais estamos lidando.
3. Fase de análise: analisamos se as propriedades são ou não satisfeitas. Em caso negativo, examinamos o contraexemplo obtido, modificamos o modelo e repetimos o procedimento.

O foco deste texto está na fase de modelagem: explicamos como modelar o sistema para verificarmos as propriedades de interesse e propomos uma lógica para especificar tais propriedades. Apresentamos também um algoritmo de Verificação de Modelos para a lógica proposta, o que faz parte da fase de execução. A fase de análise corresponde ao processo de verificação aplicado durante o processo de desenvolvimento do *software*. Consiste em corrigir *bugs* e remodelar o sistema quando necessário. O objetivo é prover uma ferramenta que possa ser utilizada pela Engenharia de *Software* para verificação formal. Como a fase de análise faz parte da utilização da ferramenta e não da sua definição e desenvolvimento, não damos ênfase a ela.

A figura a seguir, retirada de (BAIER; KATOEN, 2008) com tradução nossa, é um diagrama com a estratégia geral para a verificação por Verificação de Modelos:

Figura 13 – Esquema da Verificação de Modelos



Fonte: adaptada de (BAIER; KATOEN, 2008)

As vantagens da Verificação de Modelos são elencadas a seguir (RYAN; HUTH, 2004):

1. É uma estratégia de verificação geral, aplicável a uma enorme quantidade de aplicações, como sistemas embutidos, engenharia de *software* e *design* de *hardware*.
2. Verificação parcial: propriedades podem ser verificadas individualmente, sem a necessidade de especificar o sistema inteiro para que a verificação aconteça.
3. Caso a propriedade verificada não seja satisfeita, temos um contraexemplo, o que é importante para realizar depuração.
4. Estudos empíricos indicam que a Verificação de Modelos leva a menores tempos de desenvolvimento (BAIER; KATOEN, 2008; RYAN; HUTH, 2004).
5. Possui uma fundamentação matemática sólida: baseia-se em teoria dos grafos, estruturas de dados e lógica.

Como desvantagens, temos (RYAN; HUTH, 2004):

1. Não funciona bem para programas que lidam com grandes quantidades de dados.
2. Existem problemas relacionados à complexidade dos algoritmos de Verificação de Modelos: caso a complexidade seja muito alta, os procedimentos de verificação não são computacionalmente eficientes.
3. Verificamos um modelo do sistema, não o sistema de fato. Uma má modelagem do sistema levará a resultados imprecisos.
4. A Verificação de Modelos não torna a atividade de realizar testes no produto final, seja *software* ou *hardware*, dispensável.
5. Não garante completude, i.e, não temos garantia que todos os fluxos de execução do *software* são avaliados.
6. O número de estados necessários para representar precisamente o sistema pode crescer exponencialmente, excedendo a quantidade de memória do computador e comprometendo a eficiência computacional.
7. Modelar o sistema sucintamente e formalizar as propriedades a serem verificadas na lógica utilizada requer experiência.

Com isso, acreditamos que o leitor tenha compreendido o que é Verificação de Modelos e tenha uma base sobre como o utilizamos na verificação de sistemas. Além das duas referências já mencionadas, sugerimos também a referência (CLARKE *et al.*, 2018) caso o leitor queira mais informações a respeito do método em questão.

3.2 Estruturas de Kripke

Conforme explicamos na seção anterior, precisamos modelar o sistema de interesse para aplicarmos o método de Verificação de Modelos. Para isso, usamos estruturas de Kripke. Apresentamos nesta subseção uma definição geral de estruturas de Kripke com a finalidade de apresentá-las ao leitor. No Capítulo 4, apresentamos as definições específicas para o problema com o qual estamos lidando.

Essas estruturas são formadas por estados e transições entre esses estados. Também é possível adicionar rótulos aos estados, de modo que esses rótulos representam informações que desejamos armazenar a respeito do estado que os possui. A definição que apresentamos a seguir

deve-se a (RYAN; HUTH, 2004).

Definição 3.2.1 (Estrutura de Kripke) *Uma estrutura de Kripke é uma tupla M = (S, →, L, AP), onde:*

1. *S é um conjunto de estados.*
2. *$\rightarrow \subseteq (S \times S)$ é uma relação de transição.*
3. *L: $S \rightarrow 2^{AP}$ é uma função de rótulos.*
4. *AP é um conjunto de átomos proposicionais.*

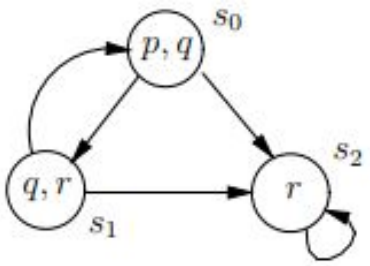
Como exemplo, temos:

Exemplo 4 *Seja M a estrutura de Kripke a seguir:*

- *$S = \{s_0, s_1, s_2\}$ são os estados.*
- *$\rightarrow = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2)\}$ é a relação de transição.*
- *$L(s_0) = \{p, q\}$, $L(s_1) = \{q, r\}$ e $L(s_2) = \{r\}$ são as atribuições de rótulos dadas pela função L.*
- *$AP = \{p, q, r\}$*

É comum utilizarmos a representação de grafos direcionados: os estados são representados por vértices, a relação de transição é representada pelas arestas e rotulamos os vértices para indicar as atribuições dadas pela função de rótulos. Caso o leitor não esteja familiarizado com grafos, que são utilizados amplamente durante todo o texto, sugerimos (WEST, 2000). A figura a seguir ilustra o exemplo dado com a representação de grafos. Voltaremos neste exemplo na seção sobre lógicas temporais.

Figura 14 – Exemplo de uma estrutura de Kripke



Fonte: (RYAN; HUTH, 2004)

A semântica das lógicas temporais é com base nas estruturas de Kripke. Para isso, o conceito de caminho é necessário. Apresentamos essa definição a seguir, de acordo com (RYAN; HUTH, 2004).

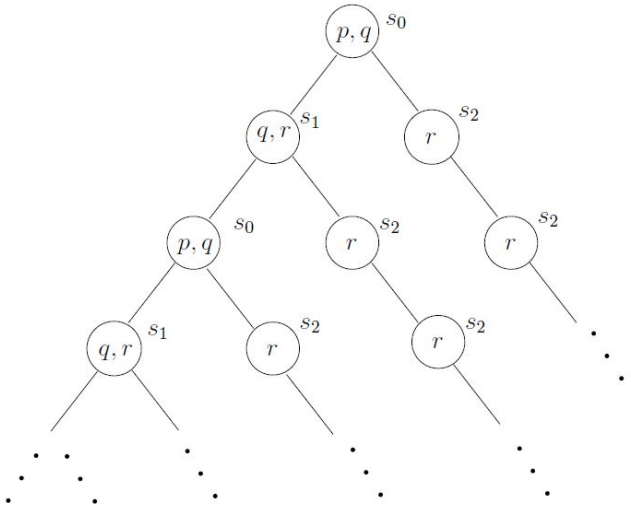
Definição 3.2.2 (Caminho em uma estrutura de Kripke) *Um caminho em uma estrutura de Kripke* $M = (S, \rightarrow, L, AP)$ *é uma sequência infinita de estados* $s_1,\ s_2,\ s_3, ... \in S$ *tal que para cada* $i \geq 1$*, temos* $s_i \rightarrow s_{i+1}$*. Escrevemos* $\pi = s_1 \rightarrow s_2 \rightarrow ...$ *para denotar um caminho. Escrevemos* π^i *para denotar o sufixo do caminho que começa no índice i, e.g,* $\pi^3 = s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow ....$

Dizemos que s_2 *é sucessor de* s_1 *se e somente se existe uma transição de* s_1 *para* s_2*, ou seja,* $s_1 \rightarrow s_2$*. Dizemos que* s_3 *é antecessor de* s_1 *se e somente se existe uma transição de* s_3 *para* s_1*, ou seja,* $s_3 \rightarrow s_1$.

Atrelada a uma estrutura de Kripke, temos também a árvore de computação, que consiste em desenrolar a estrutura a partir de um estado, que é a raiz da árvore, expondo todos os caminhos possíveis a partir desse estado. Essa árvore é infinita se e somente se a estrutura de Kripke possui pelo menos um estado que tem sempre um sucessor. Neste texto, assumimos que todo estado em uma estrutura de Kripke possui pelo menos um sucessor, pois consideramos o tempo como sendo infinito, conforme explicamos na próxima seção. Logo, as árvores de computação com as quais lidamos são infinitas e todos os seus ramos são infinitos. Cada caminho possível na estrutura a partir de um certo estado corresponde a um ramo da árvore.

A seguir, temos a árvore de computação da estrutura mostrada na Figura 14 considerando o estado s_0 como raiz:

Figura 15 – Exemplo de uma árvore de Computação



Fonte: (RYAN; HUTH, 2004)

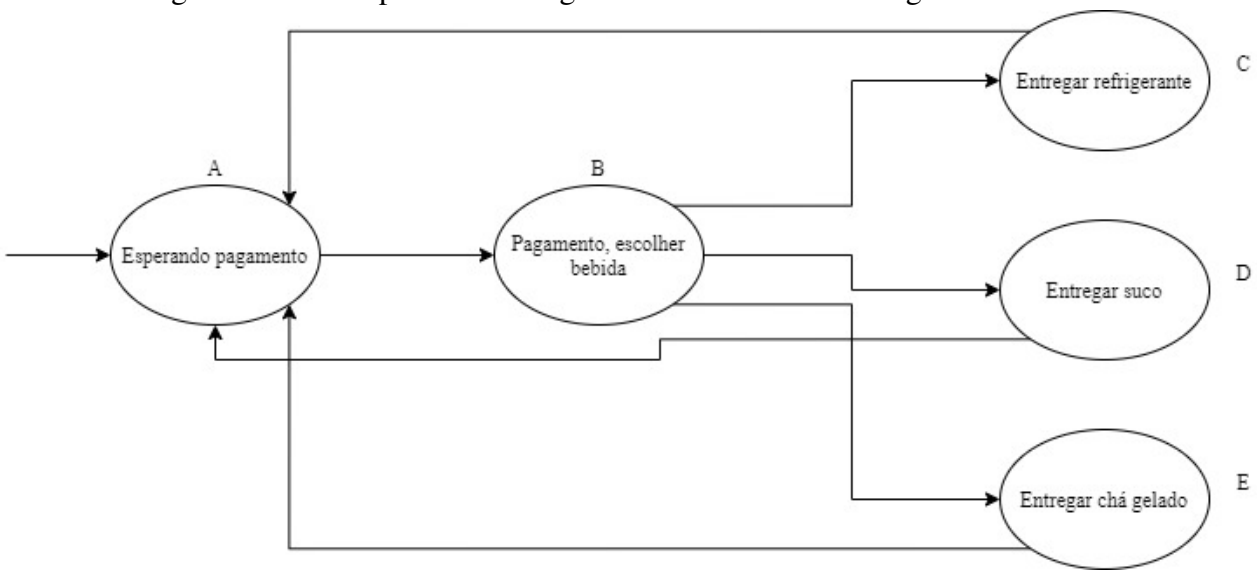
As árvores de computação facilitam a análise dos caminhos existentes em uma estrutura de Kripke, o que é bastante útil quando avaliamos fórmulas de lógicas temporais. Para

o exemplo em questão, temos s_0 como raiz da árvore. Partimos desse estado e seguimos todos os caminhos possíveis, tornando cada caminho um ramo da árvore. Em estados que possuem mais de um sucessor, criamos uma ramificação para cada sucessor. Dessa forma, é possível visualizar sucintamente todas as possibilidades de caminhos na estrutura.

Apresentamos a modelagem de um sistema muito simples utilizando estruturas de Kripke. Na próxima seção, falaremos sobre lógicas temporais e voltaremos a esse exemplo para mostrar como formalizar uma especificação dele usando uma lógica temporal.

Exemplo 5 (Sistema simplificado de uma máquina de bebidas) *Consideramos o sistema simplificado de uma máquina de venda de bebidas. A máquina em questão vende três tipos de bebidas: refrigerante, suco e chá gelado. A máquina fica em um estado de espera até receber o pagamento do usuário, quando passa para um estado de escolha. O usuário escolhe sua bebida, a máquina a entrega e retorna para o estado de espera. Nos rótulos dos estados, temos o que é verdadeiro naquele estado, e.g, no estado de escolha, o átomo proposicional pagamento é satisfeito, pois o sistema atinge esse estado apenas se o usuário inseriu dinheiro na máquina. A aresta sem origem indica o estado inicial do sistema. Esse exemplo trata-se de uma adaptação de um exemplo apresentado em (BAIER; KATOEN, 2008). Mostramos a representação de grafos direcionados a seguir:*

Figura 16 – Exemplo da modelagem de um sistema de refrigerante



Fonte: (BAIER; KATOEN, 2008) - Adaptada

Descrevendo a estrutura de Kripke, temos:

1. S = {A, B, C, D, E}
2. $\rightarrow$ = {(A, B), (B, C), (B, D), (B, E), (C, A), (D, A), (E, A)}
3. L(A) = {esperando_pagamento}, L(B) = {pagamento, escolher_bebida}, L(C) = {entregar_refrigerante}, L(D) = {entregar_suco}, L(E) = {entregar_chá_gelado}
4. AP = {esperando_pagamento, pagamento, escolher_bebida, entregar_refrigerante, entregar_suco, entregar_chá_gelado}

Como um exemplo de especificação, considere "Após realizar o pagamento, a máquina sempre entrega uma bebida ao usuário". Essa é claramente uma propriedade desejável para o sistema em questão. Observe que a mesma é satisfeita pelo modelo apresentado: no estado B, o pagamento é realizado. A partir de B, podemos ir para três estados, a saber, C, D e E. Em todos esses estados, uma das três bebidas disponíveis é entregue. Logo, após o pagamento ser realizado, a máquina libera uma bebida. Na seção seguinte, mostramos como escrever essa especificação formalmente usando uma lógica temporal.

3.3 Lógicas temporais

Conforme explicamos na Seção 3.1, precisamos formalizar as propriedades que desejamos investigar na Verificação de Modelos. Lógicas temporais são comumente usadas para essa finalidade.

Sistemas de *software* e de *hardware* são dinâmicos, de modo que uma propriedade que é válida em um dado momento pode não ser mais válida no futuro. Em outras palavras, à medida que o sistema é executado, propriedades são satisfeitas e deixam de serem satisfeitas. Lógicas proposicionais e de primeira ordem analisam o valor verdade de uma fórmula de um ponto de vista estático. Logo, tais lógicas não são adequadas para capturar os aspectos dinâmicos desses sistemas.

Lógicas temporais possuem um aspecto dinâmico: a verdade de uma fórmula depende do ponto do tempo no qual estamos. Com elas, conseguimos analisar propriedades que variam com o tempo. Ou seja, é possível verificar se uma propriedade que não é válida em um dado momento será válida no futuro ou vice-versa, analisar se uma propriedade é sempre válida durante a execução de um sistema, dentre outras. Assim, lógicas temporais são apropriadas para

realizar verificação de sistemas de *software* e de *hardware*.

Inúmeras propriedades que são desejáveis em sistemas dependem do tempo, visto como a sucessão de estados pelos quais o sistema passa durante sua execução. Por causa disso, conseguimos representá-las fácil e naturalmente com lógicas temporais.

Existem diversas lógicas temporais que são utilizadas com diferentes propósitos. Essas lógicas são organizadas de acordo com a maneira de encarar o tempo. Lógicas de tempo linear consideram o tempo como sendo um conjunto de caminhos, onde um caminho é uma sucessão de instantes no tempo. Lógicas de tempo ramificado representam o tempo como uma árvore, cuja raiz é o instante presente do tempo e as ramificações são os futuros possíveis. O tempo pode ainda ser visto como discreto ou contínuo.

As lógicas temporais LTL (*Linear Temporal Logic*) e CTL (*Computation Tree Logic*) mostraram-se extremamente úteis na verificação de *hardware* e protocolos de comunicação. Por isso, passaram a ser utilizadas na verificação de sistemas de *software* e hoje em dia dominam a área. A lógica que propomos no Capítulo 4 utiliza a lógica CTL, por isso, apresentamos a CTL nesta seção.

Estamos lidando com propriedades em cima de caminhos: às vezes propriedades relacionadas à existência de caminhos, outras vezes verificamos se propriedades são válidas em todos os caminhos. A LTL não nos permite falar, para o caso geral, sobre propriedades a respeito da existência de caminhos. Nos casos em que podemos fazer isso, precisamos utilizar a negação da fórmula que quantifica sobre todos os caminhos para falar sobre a existência de caminhos. Essa estratégia pode levar a fórmulas difíceis de serem lidas ou pouco intuitivas. Como várias das propriedades com as quais estamos lidando falam sobre propriedades relacionadas à existência de caminhos, conseguimos expressá-las melhor em CTL. Por isso, escolhemos a CTL e não a LTL.

Outra lógica temporal bastante conhecida é a CLT*. Essa lógica possui os poderes expressivos da LTL e da CTL e é mais expressiva do que ambas. Entretanto, ainda preferimos a CTL: conseguimos escrever fórmulas da CTL de maneira mais intuitiva do que em CTL* e o algoritmo de Verificação de Modelos possui uma complexidade melhor. Além disso, a CTL mostrou-se expressiva o suficiente para expressarmos as especificações de *software* que encontramos. Logo, não precisamos de uma lógica temporal mais expressiva.

Conforme explicamos no Capítulo 1, estamos trabalhando com dois níveis de verificação. Para o nível 1, que trata uma versão de *software* específica, o tempo corresponde à sucessão

de chamadas de métodos e capturamos essa noção com lógicas temporais. Para o nível 2, que lida com todas as versões do sistema sendo desenvolvido, o tempo é a própria ordenação das versões, que são desenvolvidas incrementalmente ao longo do tempo. Logo, lógicas temporais são adequadas para a análise de propriedades desenvolvidas neste trabalho.

Essa introdução sobre lógicas temporais é uma coletânea de informações de (BAIER; KATOEN, 2008; RYAN; HUTH, 2004; FISHER, 2011). Para a subseção a seguir, usamos a referência (RYAN; HUTH, 2004), porém é possível encontrar a semântica, sintaxe e exemplos da lógica temporal apresentada em (BAIER; KATOEN, 2008; Beyersdorff *et al.*, 2009). Sugerimos ainda (FISHER, 2011) para aplicações de lógicas temporais na análise formal de sistemas.

3.3.1 CTL

A CTL é uma lógica de tempo ramificado. Logo, o tempo é visto como uma árvore e cada instante, representado por um vértice da árvore, pode possuir qualquer número finito e positivo de sucessores. O tempo é considerado como sendo discreto, ou seja, o tempo é uma sucessão de pontos e não uma linha contínua.

Dependendo da aplicação para a qual estamos usando CTL, podemos considerar o tempo como sendo finito ou infinito. No caso do tempo infinito, todo vértice da árvore que representa o tempo possui pelo menos um sucessor. Dessa forma, indicamos que todo ponto no tempo é sempre seguido por outro ponto no tempo, ou seja, o tempo nunca para.

A CTL permite quantificarmos explicitamente em cima dos caminhos. Essa lógica possui os quantificadores A e E (“todos os caminhos”, “existe um caminho”, respectivamente) que operam apenas nos conectivos temporais. Por exemplo, a fórmula EFq significa que existe um caminho que em algum momento satisfaz o átomo proposicional q e representa a propriedade “Existe um estado alcançável que satisfaz q”. Explicamos os detalhes da sintaxe e da semântica da CTL a seguir.

Definição 3.3.1 (Sintaxe da CTL) *Apresentamos a sintaxe da lógica CTL usando a forma de Backus Naur:*

$$\alpha := \bot \mid \top \mid p \mid (\neg\alpha) \mid (\alpha \vee \alpha) \mid (\alpha \wedge \alpha) \mid (\alpha \rightarrow \alpha) \mid (AX\alpha) \mid (EX\alpha) \mid (AF\alpha) \mid (EF\alpha) \mid (AG\alpha) \mid (EG\alpha) \mid AU(\alpha,\alpha) \mid EU(\alpha,\alpha)$$

Exemplos de fórmulas da CTL:

1. $AG(q \rightarrow EGr)$
2. $AU(p_1, AU(p_2, p_3))$
3. $EF(EU(r, q))$

A semântica da CTL é definida com base nas estruturas de Kripke. Cada estado de uma estrutura de Kripke representa um ponto no tempo e as transições indicam os sucessores de um ponto no tempo. Usamos a função de rótulos para dar informações a respeito de um dado ponto no tempo. Neste trabalho, consideramos o tempo como sendo infinito. Na semântica da CTL, isso é codificado ao especificarmos que todo estado deve possuir pelo menos um sucessor.

Definição 3.3.2 (Semântica da CTL) *Sejam M = (S, $\rightarrow$, L, AP) uma estrutura de Kripke tal que todo estado de S possui pelo menos um estado sucessor na relação de transição, $s \in S$ e ϕ uma fórmula da CTL. A semântica da CTL é definida como segue:*

1. $M, s \models \top$
2. $M, s \not\models \bot$
3. $M, s \models p$ *sse* $p \in L(s)$
4. $M, s \models \neg\phi$ *sse* $M, s \not\models \phi$
5. $M, s \models \phi_1 \wedge \phi_2$ *sse* $M, s \models \phi_1$ *e* $M, s \models \phi_2$
6. $M, s \models \phi_1 \vee \phi_2$ *sse* $M, s \models \phi_1$ *ou* $M, s \models \phi_2$
7. *M,* $s \models \phi_1 \rightarrow \phi_2$ *sse M,* $s \models \phi_2$ *sempre que* $M, s \models \phi_1$
8. $M, s \models AX\phi$ *sse para todo* s_2 *tal que* $s \rightarrow s_2$*, temos que* $M, s_2 \models \phi$
9. $M, s \models EX\phi$ *sse existe algum* s_2 *tal que* $s \rightarrow s_2$ *e* $M, s_2 \models \phi$
10. $M, s \models AG\phi$ *sse para todo caminho* $s = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$ *e para todo* s_i *ao longo do caminho, temos que* $M, s_i \models \phi$
11. $M, s \models EG\phi$ *sse existe algum caminho* $s = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$ *e para todo* s_i *ao longo do caminho, temos que* $M, s_i \models \phi$
12. $M, s \models AF\phi$ *sse para todo caminho* $s = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$ *existir algum* s_i *ao longo do caminho tal que* $M, s_i \models \phi$
13. $M, s \models EF\phi$ *sse existe algum caminho* $s = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$ *tal que existe algum* s_i *ao longo deste caminho tal que* $M, s_i \models \phi$
14. $M, s \models AU(\phi_1, \phi_2)$ *sse para todos os caminhos* $s = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$ *temos que existe algum* $i \geq 1$ *tal que* $M, s_i \models \phi_2$ *e para todo j tal que* $1 \leq j \leq i$ *- 1, temos que* $M, s_j \models \phi_1$

15. $M, s \models EU(\phi_1, \phi_2)$ *sse existe um caminho* $s \to s_2 \to s_3 \to ...$ *tal que existe algum* $i \geq 1$ *tal que* $M, s_i \models \phi_2$ *e para todo j tal que* $1 \leq j \leq i$ *- 1, temos que* $M, s_j \models \phi_1$

Considere o Exemplo 3.2. Temos que:

1. M, $s_0 \models EX(q \wedge r)$, pois temos o caminho $s_0 \to s_1 \to s_0 \to s_1 \to ...$ e q, $r \in L(s_1)$.
2. $M, s_2 \models EGr$, pois existe o caminho $s_2 \to s_2 \to s_2 ...$ e $r \in L(s_2)$.

Na referência (RYAN; HUTH, 2004), encontramos exemplos de especificações de propriedades comuns em sistemas de *software* e suas representações em CLT. Por exemplo, considere a propriedade "Se um usuário solicita um determinado recurso A, então este usuário terá sua solicitação eventualmente atendida". A fórmula da CTL *AG(EF reiniciar)* formaliza o requisito "De qualquer estado, é possível alcançar um estado no qual é permissível reiniciar o sistema".

Exemplo 6 (Sistema da máquina de bebidas) *Voltamos para o Exemplo 3.2 Temos a propriedade "Após realizar o pagamento, a máquina sempre entrega uma bebida ao usuário". Desejamos que essa propriedade seja verdadeira para todo o sistema. Por isso, começamos a fórmula com o operador AG. A máquina deve entregar uma bebida apenas quando o pagamento for realizado. Em outras palavras, se o pagamento foi feito, então a máquina deve entregar uma bebida. Essa estrutura de se-então nos compele a utilizar uma implicação. A especificação não diz qualquer coisa sobre quando a máquina deve entregar a bebida. Por isso, consideramos que a bebida pode ser entregue a qualquer momento uma vez que o pagamento foi realizado. Representamos isso utilizando o operador AF. A fórmula resultante é:*

AG(pagamento $\to$ *AF((entregar_refrigerante)* $\vee$ *(entregar_suco)* $\vee$ *(entregar_chá_gelado)))*

Essa fórmula quer dizer "Sempre que o átomo proposicional pagamento for verdadeiro em um estado, então em algum momento do futuro, a partir desse estado, chegamos em um estado no qual um dos três átomos proposicionais que indicam a entrega de uma bebida é verdadeiro". Note que a especificação apenas pede que uma bebida seja entregue e não obriga que uma única bebida seja entregue por pagamento. Por isso, utilizamos simplesmente uma

disjunção. Observe que o modelo em questão satisfaz à fórmula dada: a explicação é análoga à que apresentamos na especificação do Exemplo 3.2, porém agora aplicamos a semântica da CTL.

Considere uma fórmula com a estrutura que apresentamos acima: $AG(p \rightarrow AF(q))$. Para essa fórmula ser satisfeita em uma estrutura de Kripke M, todo estado da estrutura deve satisfazer $p \rightarrow AF(q))$, conforme a semântica do conectivo AG. Pela semântica da implicação, se um estado v satisfaz o átomo proposicional p, então v deve satisfazer $AF(q)$, ou seja, em todos os caminhos que começam em v, temos um estado que satisfaz q. E esse padrão deve se repetir indefinidamente pelo caminho: suponha que temos três estados v_1, v_2 e v_3 que aparecem nesta ordem ao longo de um caminho. Temos que v_1, v_3 satisfazem p e v_2 satisfaz q. Os demais estados desse caminho não satisfazem p ou q. Logo, $v_1 \models p \rightarrow AF(q)$, conforme o que acabamos de explicar. Porém, $v_3 \not\models p \rightarrow AF(q)$, pois não existe um estado após v_3 que satisfaz q. Lembre-se que a CTL é uma lógica de tempo ramificado. Esse é um comportamento da semântica dessas lógicas. Em lógicas de tempo linear, como já temos o estado v_2 satisfazendo q, isso seria o suficiente para v_3 satisfazer a fórmula mencionada, mesmo que v_2 esteja antes de v_3 no caminho. Essas lógicas avaliam as fórmulas em caminhos e não em estados.

Agora que já conhecemos a sintaxe e a semântica da CTL, explicamos a seguir um algoritmo de Verificação de Modelos para essa lógica, ou seja, um algoritmo que resolve a Verificação de Modelos quando usamos a lógica CTL para formalizar as propriedades de interesse.

3.4 Algoritmo de Verificação de Modelos para a CTL

Nesta seção, explicamos um algoritmo de Verificação de Modelos para a lógica CTL. Sejam M uma estrutura de Kripke, s um estado de M e α uma fórmula da CTL. Esse algoritmo almeja responder à seguinte pergunta:

A estrutura M, a partir do estado s, satisfaz a fórmula α?

Denotamos essa pergunta por $M, s \models \alpha$.

Podemos considerar esse problema de vários modos. Por exemplo, podemos ter

como entrada M, s e α e responder simplesmente sim ou não. Ou seja, a estrutura M tendo s como estado inicial satisfaz ou não a fórmula α. Ou então, consideramos a entrada como sendo somente M e α e a saída corresponde a todos os estados que satisfazem a fórmula α. Note que essa segunda abordagem resolve também a primeira, pois é suficiente verificar se o estado s está entre os estados que satisfazem α. O algoritmo que apresentamos para a CTL segue essa estratégia.

3.4.1 Algoritmo para a CTL

O algoritmo que apresentamos aqui pode ser encontrado em (RYAN; HUTH, 2004) e vamos chamá-lo de algoritmo de marcação. A ideia geral do algoritmo é percorrer a estrutura de Kripke marcando quais subfórmulas da fórmula de entrada são satisfeitas em cada estado de acordo com regras relacionadas à semântica da CTL. Ao final do processo, temos os estados que satisfazem à fórmula de entrada. Para sintetizar, temos:

1. Entrada: estrutura de Kripke M e fórmula da CTL α.
2. Saída: todos os estados marcados com quais subfórmulas de α eles satisfazem, incluindo a própria α.

O conjunto $\{\bot, \neg, \wedge, AF, EU, EX\}$ é um conjunto adequado de conectivos da CTL (RYAN; HUTH, 2004). O algoritmo utiliza esse conjunto. Com isso, temos menos regras para decidir se um estado satisfaz ou não a uma subfórmula, facilitando o procedimento. O primeiro passo do algoritmo é, portanto, reescrever α utilizando somente esses conectivos.

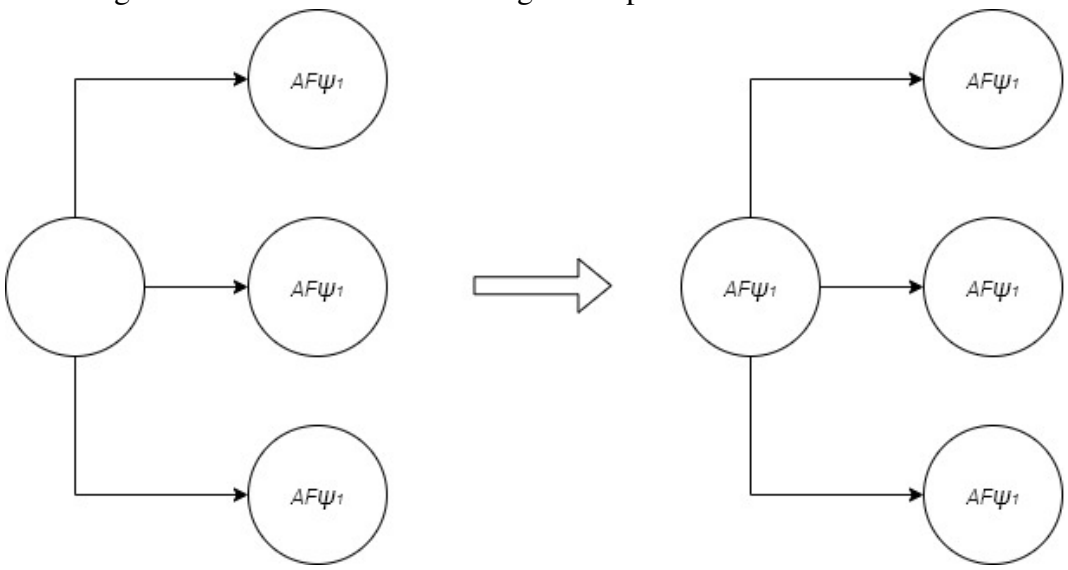
Um estado é marcado com uma subfórmula ψ de α se esse estado satisfaz ψ. O procedimento começa marcando os estados a partir das menores subfórmulas de α, que são os átomos proposicionais, e segue para subfórmulas maiores, i.e, com mais conectivos, até chegar na fórmula α. As regras de marcação são descritas a seguir.

1. Seja ψ uma subfórmula de α, s um estado qualquer de M e assuma que os estados satisfazendo as subfórmulas próprias de ψ já foram marcados.
2. Marcamos com ψ os estados que satisfazem às regras a seguir. Se ψ é:
   - $\bot$: nenhum estado é marcado com $\bot$, pois $\bot$ nunca é satisfeito.
   - Átomo proposicional p: s é marcado com p se e somente se $p \in L(s)$.
   - $\psi_1 \wedge \psi_2$: s é marcado com $\psi_1 \wedge \psi_2$ se e somente se s já foi marcado com ψ_1 e

ψ_2.

- $\neg\psi_1$: s é rotulado com $\neg\psi_1$ se e somente se s não foi marcado com ψ_1.
- $EX(\psi_1)$: s é rotulado com $EX(\psi_1)$ se e somente se algum dos seus sucessores, i.e, s_1 tal que $s \rightarrow s_1$, foi rotulado com ψ_1.
- $AF(\psi_1)$: se s está marcado com ψ_1, marcamos-o com $AF(\psi_1)$. Marcamos com $AF(\psi_1)$ qualquer estado se todos os seus sucessores estão marcados com $AF(\psi_1)$ até que não ocorram mais mudanças. Ilustramos esse passo a seguir:

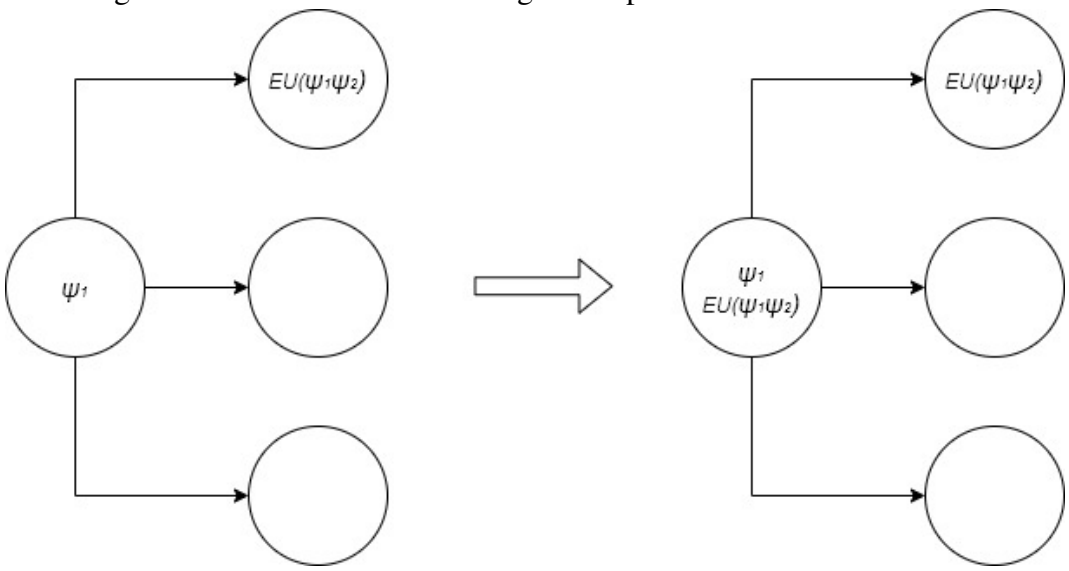
Figura 17 – Funcionamento do algoritmo para o conectivo AF



Fonte: (RYAN; HUTH, 2004)

- EU(ψ_1, ψ_2): se s está marcado com ψ_2, marcamos-o com EU(ψ_1, ψ_2). Marcamos com EU(ψ_1, ψ_2) qualquer estado que esteja marcado com ψ_1 e tem pelo menos um dos seus sucessores marcado com EU(ψ_1, ψ_2) até que não ocorram mais mudanças. Ilustramos esse passo a seguir:

Figura 18 – Funcionamento do algoritmo para o conectivo EU



Fonte: (RYAN; HUTH, 2004)

A corretude do algoritmo pode ser encontrada em (RYAN; HUTH, 2004). Apresentamos as complexidades dos algoritmos usando notação O. Caso o leitor não esteja familiarizado com ela, sugerimos consultar (CORMEN *et al.*, 2009). Considere:

1. n é o número de estados da estrutura de Kripke.
2. m é o número de transições na estrutura.
3. t é tamanho da fórmula, i.e, o número de conectivos nela.

Para verificar os átomos proposicionais, percorremos o conjunto correspondente à função L de um estado. O custo disso é $O(|AP|)$, ou seja, é o tamanho do conjunto de átomos proposicionais. Para os conectivos booleanos, verificamos a lista de subfórmulas que são satisfeitas pelo estado. Temos custo linear no tamanho da fórmula. Esses custos são dominados pelos custos descritos a seguir.

Percorrer uma estrutura de Kripke, que é um grafo com rótulo nos vértices, tem custo $O(n + m)$ (CORMEN *et al.*, 2009): podemos usar uma busca em largura, por exemplo. Para cada conectivo temporal em α, percorremos a estrutura de Kripke. Com isso, temos custo $O(t \times (n + m))$. Quando lidamos com os conectivos temporais AF e EU, repetimos as regras f e g, respectivamente, até que não ocorram mais mudanças. No pior caso, visitamos todos os estados da estrutura. Logo, o custo do algoritmo é $O(t \times n \times (n + m))$.

3.5 Lógicas híbridas

Lógicas modais, como a CTL que apresentamos anteriormente, operam com base nas estruturas de Kripke (ou algum outro tipo de estrutura semelhante) e os estados possuem um papel fundamental nisso. Entretanto, os estados não são refletidos na sintaxe dessas lógicas: não existe controle sobre os estados. No máximo, podemos escolher o estado inicial a partir do qual avaliamos uma fórmula em uma estrutura de Kripke. As modalidades, definidas pela semântica, encarregam-se de verificar quais são estados acessíveis ao percorrer a estrutura. Entretanto, não temos controle sobre isso.

Pode ser que a aplicação para a qual estamos usando lógicas modais necessite controlar melhor os estados. Por exemplo, pode ser preciso que a sintaxe da lógica tenha uma maneira de nomear estados ou um mecanismo para acessar diretamente um estado. Para lidar com esses problemas, temos as lógicas híbridas (BLACKBURN *et al.*, 2002).

As lógicas híbridas utilizam um tipo especial de átomos proposicionais, os chamados nominais, para nomear os estados. Cada nominal está associado a um único estado. Dessa forma, podemos nos referir diretamente a um estado usando nominais. Apresentamos a seguir a sintaxe e a semântica de uma lógica híbrida com a finalidade de apresentar os operadores que interessam a este trabalho.

Definição 3.5.1 (Sintaxe da lógica híbrida) *Seja AP = {p, q, r...} um conjunto de átomos proposicionais e I = {i, j, k...} um conjunto não vazio de nominais. Temos que AP e I são conjuntos disjuntos. Definimos a sintaxe da lógica híbrida usando a notação de Backus-Naur:*

$$\alpha = p \mid i \mid \bot \mid \neg\alpha \mid \alpha \wedge \alpha \mid \diamond\alpha \mid @_i\alpha$$

Para a semântica, temos:

Definição 3.5.2 *Seja M uma estrutura de Kripke, s um estado de M, α_1 e α_2 fórmulas e N: I $\rightarrow$ S uma função que associa os nominais aos estados correspondentes. A semântica da lógica híbrida é definida a seguir:*

1. $M, s \models p$ *sse* $p \in L(s)$.
2. $M, s \models i$ *sse* $N(i) = s$.
3. $M, s \not\models \bot$.
4. $M, s \models \neg\alpha_1$ *sse* $M, s \not\models \alpha_1$.

5. $M, s \models \alpha_1 \wedge \alpha_2$ *sse* $M, s \models \alpha_1$ *e* $M, s \models \alpha_2$.
6. $M, s \models \diamond\alpha_1$ *sse existe algum* s_1 *tal que* $s \rightarrow s_1$ *e* $s_1 \models \alpha_1$.
7. $M, s \models @_i\alpha_1$ *sse* $M, v \models \alpha_1$, *onde* $N(i) = v$.

Dos conectivos apresentados, os únicos diferentes dos conectivos da CTL são os nominais e o operador $@_i$. Note que o conectivo $\diamond$ possui a mesma semântica do conectivo EX. No caso dos nominais, a semântica é muito semelhante à semântica dos átomos proposicionais: no lugar de verificar se o átomo pertence ao rótulo do estado, verificamos se o nominal está associado ao estado por meio da função N.

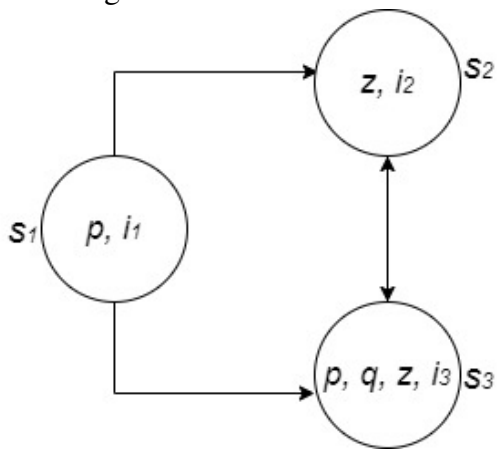
O conectivo $@_i$ possui o seguinte significado (BLACKBURN *et al.*, 2002): $@_i\alpha$ é verdadeira em um estado se e somente se α é verdadeira no único estado nomeado pelo nominal i. Com esse conectivo, podemos acessar diretamente um estado: temos os nominais para identificar cada estado e com o conectivo $@_i$, podemos analisar se o estado identificado por i satisfaz uma determinada fórmula, o que nos dá controle sobre os estados da estrutura de Kripke. Esse é o mecanismo de acesso direto aos estados que mencionamos nos primeiros parágrafos desta seção.

Exemplo 7 (Exemplos de fórmulas da lógica híbrida) *Considere as seguintes fórmulas da lógica híbrida que apresentamos:*

1. $\alpha := @_{i_3}(p \wedge q)$
2. $\beta := \diamond(i_3)$
3. $\delta := \diamond(p \wedge q)$

Considere a seguinte estrutura de Kripke M:

Figura 19 – Estrutura de Kripke para ilustrar as lógicas híbridas



Fonte: autoria própria (2021)

Analisamos a satisfatibilidade das fórmulas listadas acima em M:

1. $M, s_1 \models \alpha$*: por causa do conectivo @, devemos analisar se estado que satisfaz o nominal* i_3 *satisfaz a fórmula no escopo desse conectivo. Esse estado é* s_3*. Como* $p, q \in L(s_3)$*, temos que* $s_3 \models p \wedge q$*.*
2. $M, s_2 \models \beta$*: temos que* $s_2 \rightarrow s_3$ *e* $s_3 \models i_3$*. Logo, o conectivo* $\diamond$ *é satisfeito.*
3. $M, s_3 \not\models \delta$*: não existe um estado s tal que* $s_3 \rightarrow s$ *e* $s \models p \wedge q$*.*

Existem lógicas híbridas que possuem outros conectivos, porém não temos necessidade desses conectivos neste trabalho: o nosso interesse é no conectivo $@_i$, conforme explicamos no Capítulo 4. Apresentamos na subseção a seguir o algoritmo de verificação para a lógica híbrida (definição 3.5) que acabamos de explicar. A partir de agora, vamos nos referir a essa lógica como lógica híbrida básica.

3.6 Algoritmo de Verificação de Modelos para a lógica híbrida básica

O algoritmo de Verificação de Modelos para a lógica híbrida básica que apresentamos repete o algoritmo para a CTL. É um algoritmo simples que realiza buscas na estrutura de Kripke e estamos apresentando-o aqui como uma sugestão nossa. Entretanto, é realmente um algoritmo básico, apenas um reaproveitamento do algoritmo para a CTL, não sendo, portanto, uma contribuição relevante deste trabalho. Adicionamos o tratamento para o conectivo $@_i$ e para os nominais. Para os átomos proposicionais, $\perp$, negação e conjunção, o algoritmo executa os mesmos passos do algoritmo de Verificação de Modelos para a CTL. Para o conectivo $\diamond$, como esse tem a mesma semântica do conectivo EX, executamos o passo do algoritmo da CTL para o

conectivo EX. Para os nominais, é suficiente verificar o resultado da função N, algo semelhante ao que é feito para o átomos proposicionais. O conectivo $@_i$ procura o estado v tal que $N(i) = v$. Logo, é suficiente realizar uma busca na estrutura de Kripke. Considere:

1. n é o número de estados da estrutura de Kripke.
2. m é o número de transições na estrutura.
3. t é o tamanho da fórmula, i.e, o número de conectivos $@_i$, $\diamond$, $\neg$, $\wedge$ na fórmula.

Para cada conectivo $@_i$ na fórmula, realizamos uma busca na estrutura de Kripke. Verificar os conectivos $\diamond$ envolve acessar os vizinhos de um estado. No pior caso, precisamos verificar a estrutura inteira: considere um estado que tem todos os estados como vizinhos. Logo, temos o custo de percorrer a estrutura inteira. Já vimos que esse custo é $O(n + m)$. Logo, o custo dessas operações é $O(t \times (n + m))$. Para verificar os átomos proposicionais, percorremos o conjunto correspondente à função L de um estado. O custo disso é $O(|AP|)$, ou seja, é o tamanho do conjunto de átomos proposicionais. Logo, teríamos $O(t \times (n + m) + |AP|)$. Entretanto, a não ser que o conjunto de átomos proposicionais seja muito grande e os estados estejam rotulados com muitos átomos, essa complexidade é dominada por $O(t \times (n + m))$. Para os nominais, o custo é constante: é suficiente verificar o resultado de uma função. Para negação e para a conjunção, temos que verificar a lista de subfórmulas que já foram satisfeitas por um dado estado, o que tem custo linear no tamanho da fórmula. Esse custo é dominado por $t \times (n + m)$. Consequentemente, a complexidade do algoritmo de Verificação de Modelos para a lógica híbrida básica é O($t \times (n + m)$).

Encerramos a apresentação sobre lógicas híbridas. O maior interesse deste trabalho é no conectivo $@_i$, conforme explicamos no capítulo a seguir.

4 LÓGICA TEMPORAL DE DOIS NÍVEIS PARA VERIFICAÇÃO DE PROPRIEDADES

Neste capítulo, explicamos a metodologia proposta por este trabalho. Como modelo para a Verificação de Modelos, usamos grafos de versões e grafos de chamadas, definidos no Capítulo 1. Para uniformizar a nossa abordagem, conectamos essas duas estruturas utilizando uma função, conforme explicado na Seção 4.2. Em seguida, apresentamos a lógica temporal proposta pelo trabalho: essa lógica possui conectivos temporais, conectivos das lógicas híbridas e um conectivo que permite a comunicação entre os dois níveis de verificação dos quais falamos sobre no Capítulo 1.

Lidamos com dois níveis de verificação:

Nível 1: trata da verificação de propriedades relacionadas às chamadas de métodos dos programas de uma versão específica de um *software*. Neste nível, grafos de chamadas são utilizados. Usamos o índice *in* para nos referimos a esse nível. Por exemplo, V_{in} é conjunto de estados da estrutura de Kripke que representa um grafo de chamadas, conforme apresentado na Seção 4.1.

Nível 2: lida com a verificação de propriedades ao longo de todo o processo de desenvolvimento de um *software*. Grafos de versões são utilizados para isso. Para nos referir a esse nível, usamos o índice *out*. Logo, V_{out} é o conjunto de estados da estrutura de Kripke que representa o grafo de versões.

As metodologias atuais de Análise de Conformidade Arquitetural lidam apenas com o nível 1 de verificação: o interesse é apenas em uma versão em particular. A nossa metodologia considera o panorama global de desenvolvimento, permitindo que especificações e propriedades sejam analisadas ao longo de todo o processo de desenvolvimento de maneira uniforme. Por meio da estrutura que descrevemos na Seção 4.2 e do conectivo lógico apresentado na Seção 4.3, obtemos essa uniformidade de tratamento.

De acordo com o que explicamos no Capítulo 3, a Verificação de Modelos precisa de um modelo do sistema, de uma linguagem formal para formalizar as especificações e propriedades que devem ser verificadas e de um algoritmo para executar se o sistema satisfaz as especificações. A seguir, explanamos como modelar o sistema, a linguagem formal utilizada e apresentamos um

algoritmo para verificar se o modelo satisfaz às fórmulas dessa linguagem.

4.1 Grafos de chamadas como estruturas de Kripke

Explicamos o que são grafos de chamadas. No Capítulo 1, apresentamos as estruturas de Kripke, que são usadas para modelar o sistema na Verificação de Modelos. Agora, unimos essas duas definições e definimos grafos de chamadas como estruturas de Kripke, ou seja, explicamos como representar grafos de chamadas por meio de estruturas de Kripke. Essa é a modelagem usada neste trabalho.

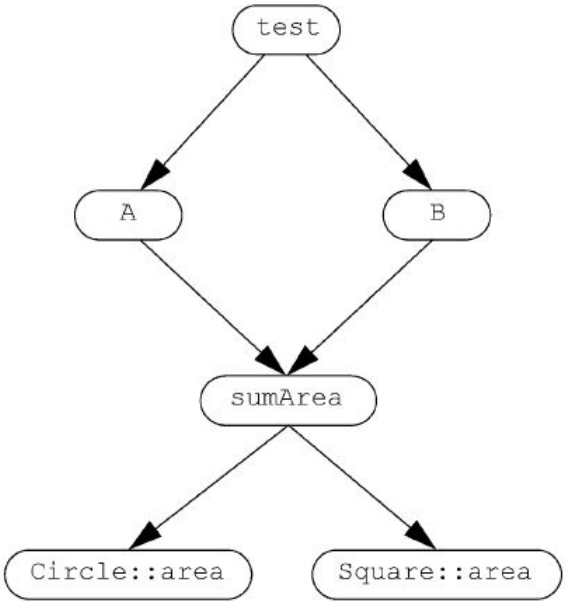
Definição 4.1.1 (Grafos de chamadas como estruturas de Kripke) *Seja* $M_{in} = (V_{in}, \rightarrow_{in}, L_{in}, PROP_{in}, v_{0in})$ *uma estrutura de Kripke, onde:*

1. V_{in} *é o conjunto de estados: representam os métodos presentes no software, ou seja, corresponde ao conjunto de vértices do grafo de chamadas.*
2. $\rightarrow_{in}$ *é o conjunto de transições: existe uma aresta de um estado A para um estado B se o método representado por A chama o método representado por B. É o conjunto de arestas do grafo de chamadas.*
3. L_{in} *é a função de rótulos:* $L_{in} : V_{in} \rightarrow 2^{PROP_{in}}$*: rotulamos os estados com o método que esse estado representa e com a classe que possui o método.*
4. $PROP_{in}$ *é o conjunto de átomos proposicionais: temos um átomo proposicional para cada método e para cada classe.*
5. v_{0in} *é o estado inicial: ao aplicarmos a Verificação de Modelos, consideramos este estado.*

Atente que o grafo de chamadas trata-se de um grafo arbitrário, não temos qualquer característica especial a respeito dele. Logo, a estrutura de Kripke resultante também será arbitrária.

Exemplo 8 (Grafo de chamadas como uma estrutura de Kripke) *Considere novamente o grafo de chamadas mostrado na Figura 3:*

Figura 20 – Grafo de chamadas



Fonte: (GROVE; CHAMBERS, 2001)

Representando o grafo acima como uma estrutura de Kripke, temos:

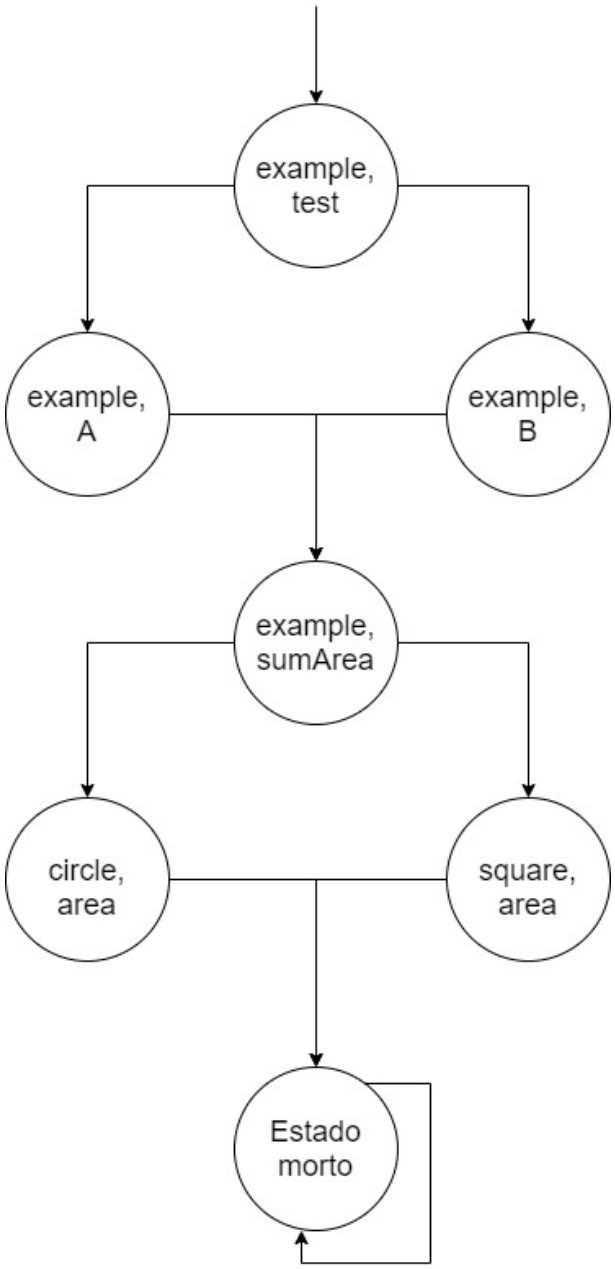
1. $V_{in} = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$
2. $\rightarrow_{in}$ = *{*(v_1, v_2)*,* (v_1, v_3)*,* (v_2, v_4)*,* (v_3, v_4)*,* (v_4, v_5)*,* (v_4, v_6)*,* (v_5, v_7)*,* (v_6, v_7)*,* (v_7, v_7)*}*
3. *Para a função de rótulos, temos:*
   - $L_{in}(v_1)$ = *{example, test};*
   - $L_{in}(v_2)$ = *{example, A};*
   - $L_{in}(v_3)$ = *{example, B};*
   - $L_{in}(v_4)$ = *{example, sumArea};*
   - $L_{in}(v_5)$ = *{circle, area};*
   - $L_{in}(v_6)$ = *{square, area};*
   - $L_{in}(v_7)$ = *{Estado_morto};*
4. $PROP_{in}$ = *{example, test, A, B, sumArea, circle, square, area, Estado_morto}*
5. $v_{0in} = v_1$

Assim como fizemos no Capítulo 3, na semântica da CTL, consideramos que todo estado possui pelo menos um sucessor, já que consideramos o tempo como sendo infinito. Para

evitar modificar a semântica da lógica, repetimos essa característica aqui. Para isso, criamos um estado morto para ser sucessor de todo estado que não possua um. Usamos um átomo proposicional para indicar qual é o estado morto.

Representando o exemplo acima como um grafo direcionado, temos:

Figura 21 – Exemplo de um grafo de chamadas como uma estrutura de Kripke



Fonte: elaborado pelo autor (2021).

Para indicar claramente o que é classe e o que é método, podemos usar a notação classe::nome_da_classe e método::nome_do_método ao criar os átomos proposicionais ou alguma outra notação que deixe explícito o que é classe o que é método. Em caso de sobrecarga de métodos (BRUEGGE; DUTOIT, 2009; DEITEL; DEITEL, 2014), onde temos vários métodos com o mesmo nome na mesma classe, podemos usar átomos proposicionais diferentes para distingui-los. Para ilustrar, considere a sobrecarga do método soma da classe Calculadora apresentada a seguir em uma sintaxe como a da linguagem C++:

1. int Calculadora::soma(int x, int y);
2. float Calculadora::soma(float x, float y);

Podemos identificar o primeiro método, que recebe inteiros, por meio dos átomos proposicionais *Calculadora, soma, int*. Para o segundo método, que lida com pontos flutuantes, usamos os átomos *Calculadora, soma, float*. Outras notações também podem ser usadas, é claro. O que queremos salientar é que átomos proposicionais podem ser usados para distinguir os métodos sobrecarregados.

Sistemas de *software* possuem inúmeras classes e métodos e seguem diversos fluxos de execução. Na prática, um sistema possui vários grafos de chamadas. Cabe aos engenheiros de *software* selecionar os trechos do código onde o método de verificação será aplicado e selecionar o vértice inicial de acordo. Não precisamos tratar desses detalhes no texto.

4.2 Grafos de versões como estruturas de Kripke

Falamos sobre grafos de versões no Capítulo 1, na Definição 1.2.2. Assim como fizemos na seção anterior, explicamos agora a nossa representação de grafos de versões como estruturas de Kripke.

Definição 4.2.1 (Grafo de versões como uma estrutura de Kripke) *Seja* $M_{out} = (V_{out}, \rightarrow_{out}, L_{out}, PROP_{out}, v_{0out}, I, GC, N, C)$ *uma estrutura de Kripke, onde:*

1. V_{out} *é o conjunto de estados do grafo de versões: cada estado representa uma versão do software.*
2. $\rightarrow_{out}$ *é o conjunto de arestas do grafo de versões: indica quais versões são as sucessoras de uma versão.*
3. L_{out} *é uma função de rótulos de átomos proposicionais:* $L_{out} : V_{out} \rightarrow 2^{PROP_{out}}$*: podem ser usados para codificar informações adicionais que sejam de interesse dos*

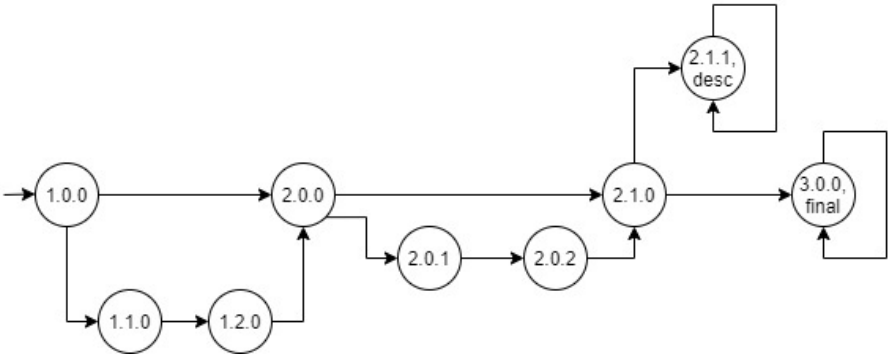
engenheiros de software a respeito de uma versão.

4. $PROP_{out}$ *é o conjunto de átomos proposicionais para o grafo de versões.*
5. v_{0out} *é o estado inicial, correspondente à primeira versão.*
6. *I é o conjunto de nominais usados para indicar as versões: neste trabalho, estamos usando a notação* $i_{n.k.j}$*, onde* $n,k,j \in \mathbb{N}$.
7. *GC é o conjunto de grafos de chamadas.*
8. *N é uma função que associa nominais aos vértices correspondentes:* $N : I \rightarrow V_{out}$.
9. *C é uma função que associa os vértices do grafo de versões ao grafo de chamadas correspondente a versão representada pelo vértice :* $C : V_{out} \rightarrow GC$.

Conforme dissermos na Subseção 1.2.2, os grafos de versões não possuem ciclos, já que não é possível uma versão lançada em um dia x ser sucedida por uma versão lançada em um dia anterior a x. Logo, podemos percorrer o grafo de chamadas apenas em uma direção, não é possível voltar. Temos apenas uma versão inicial e uma versão final. Assumimos que a partir da versão inicial, podemos alcançar todas as demais versões, já que elas são derivadas da versão inicial. A estrutura de Kripke resultante preserva essas características. Temos ainda que cada caminho na estrutura representa um ramo de desenvolvimento. Podemos mudar o estado inicial à vontade para analisar diferentes ramos (*branch*) de desenvolvimento.

Da mesma maneira que fizemos para os grafos de chamadas, para não precisarmos alterar a semântica da CTL, adicionamos um laço no estado correspondente à versão final. Em caso de atualização do *software*, esse laço é removido e uma aresta para a próxima versão é adicionada. Também realizamos esse procedimento para versões intermediárias que sejam descartadas durante o processo de desenvolvimento. Ilustramos isso na figura a seguir:

Figura 22 – Grafo de versões como uma estrutura de Kripke com laços



Fonte: elaborado pelo autor (2021).

De fato, a estrutura de Kripke resultante do grafo de chamadas não possui ciclos. Entretanto, com a modificação que acabamos de fazer, passamos a ter ciclos, porém esses ciclos são laços e a característica de não ser possível voltar para versões anteriores continua válida. Para indicar a versão inicial, usamos uma seta. Para indicar as versões finais e descartadas, podemos usar átomos proposicionais, conforme ilustramos na figura acima: a versão 3.0.0 é a versão final, rotulada com o átomo final. Já a versão 2.1.1 é uma versão descartada durante o processo de desenvolvimento, por isso a rotulamos com desc.

Para identificar as versões, usamos a notação $i_{n.k.j}$, onde $n.k.j \in \mathbb{N}$. A versão inicial é identificada pelo nominal $i_{1.0.0}$. Para o caso no qual $k = j = 0$, abreviamos a notação para i_n. Logo, a primeira versão é identificada por i_1 e a terceira versão, por exemplo, pelo nominal i_5. Se estamos na versão $i_{n.k.j}$, a próxima versão é identificada por meio do incremento de um desses índices de acordo com a ideia de versionamento semântico explicada na Subseção 1.2.2. Apresentamos um exemplo no final desta seção. A função N nos diz qual estado está associado a um certo nominal. Por se tratar de uma função, cada nominal está associado a apenas um estado. Entretanto, podemos ter um estado associado a mais de um nominal. Isso não representa problema algum para a metodologia que estamos propondo e pode ser facilmente tratado restringindo N a uma função bijetiva.

Temos um conjunto *GC* de grafos de chamadas e cada versão possui o seu grafo de chamadas correspondente. Para indicar qual é o grafo de chamadas de uma versão, usamos a função *C*: por exemplo, $C(v)$ para $v \in V_{out}$ é o grafo de chamadas correspondente ao estado v do grafo de versões. Novamente, por tratar-se de uma função, cada grafo de chamadas está associado a apenas uma versão.

Observe que nos elementos que são comuns para as estruturas dos grafos de chamadas e grafos de versões, usamos os índices *in* e *out* para diferenciar o que é de cada estrutura. Já para os elementos exclusivos das estruturas dos grafos de versões, removemos o índice.

Na próxima seção, apresentamos a lógica proposta neste trabalho. A semântica dessa lógica está definida com base no grafo de versões como estruturas de Kripke que acabamos de definir. No restante deste capítulo, vamos nos referir às estruturas de Kripke que representam grafos de chamadas ou grafos de versões como apenas grafos de chamadas ou grafos de versões a fim de simplificar a escrita.

4.3 Lógica proposta

Propomos uma variação da lógica temporal CTL que utiliza operadores de lógicas híbridas. Chamamos essa lógica de lógica temporal de dois níveis. As fórmulas dessa lógica são constituídas de dois níveis: um nível é para verificar propriedades em cima do grafo de chamadas (nível 1) e o outro, para verificar propriedades em cima do grafo de versões (nível 2). Conforme falamos na introdução deste capítulo, usamos os índices *in* e *out* para nos referirmos a cada um desses níveis. A comunicação entre esses dois níveis é realizada pelo conectivo IN. Explicamos mais detalhes desse conectivo após apresentar a sintaxe e a semântica.

No Capítulo 3, apresentamos a definição de caminho em uma estrutura de Kripke (Definição 3.2). Essa definição também se aplica às estruturas que definimos nas duas seções anteriores: um caminho em uma estrutura de Kripke $M = (V, \rightarrow, L, PROP, v_0)$ é uma sequência infinita de estados $v_1,\ v_2,\ v_3, ... \in V$ tal que para cada $i \geq 1$, temos $v_i \rightarrow v_{i+1}$. Escrevemos $\pi = v_1 \rightarrow v_2 \rightarrow ...$ para denotar um caminho. Escrevemos π^i para denotar o sufixo do caminho que começa no índice i, e.g, $\pi^3 = v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow ....$ Não importa se estamos lidando com grafos de chamadas ou grafos de versões: como para a definição de caminho apenas os estados V e as transições $\rightarrow$ importam, essa definição serve para as duas estruturas. Agora podemos mostrar as definições da sintaxe e da semântica da lógica que propomos neste trabalho.

Definição 4.3.1 (Sintaxe da lógica proposta) *Apresentamos a sintaxe da lógica proposta usando a forma de Backus-Naur:*

1. *Para as fórmulas que são avaliadas sobre grafos de chamadas:*
   $\alpha_{in} := p_{in} \| \neg\alpha_{in} \| \alpha_{in} \vee \alpha_{in} \| \alpha_{in} \wedge \alpha_{in} \| \alpha_{in} \rightarrow \alpha_{in} \| AX\alpha_{in} \| EX\alpha_{in} \| AF\alpha_{in} \| EF\alpha_{in} \| AG\alpha_{in} \| EG\alpha_{in} \| AU(\alpha_{in}, \alpha_{in}) \| EU(\alpha_{in}, \alpha_{in}) \| AP(\alpha_{in}) \| EP(\alpha_{in})$
2. *Para as fórmulas que são avaliadas sobre o grafo de versões:*
   $\alpha_{out} := p_{out} \| i \| \neg\alpha_{out} \| \alpha_{out} \vee \alpha_{out} \| \alpha_{out} \wedge \alpha_{out} \| \alpha_{out} \rightarrow \alpha_{out} \| AX\alpha_{out} \| EX\alpha_{out} \| AF\alpha_{out} \| EF\alpha_{out} \| AG\alpha_{out} \| EG\alpha_{out} \| AU(\alpha_{out}, \alpha_{out}) \| EU(\alpha_{out}, \alpha_{out}) \| @_i\alpha_{out} \| IN\alpha_{in}$

Exemplo 9 *Como a estrutura de Kripke dos grafos de versões é a estrutura base do nosso método de Verificação de Modelos, construímos as fórmulas a partir da cláusula 2 da definição da lógica proposta:*

1. $\alpha_1 := p \wedge i$, *para* $p \in PROP_{out}$ *e* $i \in I$*: essa fórmula verifica se um estado satisfaz um átomo proposicional p e é identificado pelo nominal i.*
2. $\alpha_2 := EF(p) \wedge @_{i_5} EX(q)$*: existe pelo menos um estado que satisfaz p e pelo menos uma versão sucessora da quinta satisfaz q.*
3. $\alpha_3 := IN(AG(s))$*: o grafo de chamadas da primeira versão satisfaz s em todos os seus estados. Observe o uso do IN para transferir a verificação para os grafos de chamadas, transacionando entre os dois níveis.*
4. $\alpha_4 := IN((s \rightarrow t) \wedge AF(z))$*: novamente, verificamos uma conjunção de fórmulas no grafo de chamadas correspondente à primeira versão.*
5. $\alpha_5 := @_{i_3} EX(p) \vee IN(EX(z))$*: verificamos se existe uma versão sucessora da versão 3 que satisfaz p ou então se o grafo de chamadas da primeira versão satisfaz* $EX(z)$*, ou seja, se o primeiro estado do grafo de chamadas possui um sucessor que satisfaz z.*

Explicamos um pouco sobre o significado das fórmulas para apresentar a intuição por atrás delas. Esses detalhes ficarão mais claros quando apresentamos a semântica a seguir. Por enquanto, queríamos apenas apresentar a construção de fórmulas usando a sintaxe proposta.

As fórmulas apresentadas acima devem ser verificadas na estrutura de Kripke M_{out} que codifica o grafo de versões conforme a definição apresentada na Seção 4.2. Consideramos que começamos a verificação a partir do estado inicial $v_{0_{out}}$:

$M_{out} \models \alpha_{out}$ se e somente se $M_{out}, v_{0_{out}} \models \alpha_{out}$, conforme a definição da semântica apresentada a seguir.

Definição 4.3.2 (Semântica da lógica proposta) *Seja* $M_{out} = (V_{out}, \rightarrow_{out}, L_{out}, PROP_{out}, v_{0_{out}}, I, GC, N, C)$ *um grafo de versões,* v_{out} *um estado desse grafo,* α_{in} *e* α_{out} *fórmulas da lógica temporal proposta. A semântica da lógica proposta é definida como segue:*

1. $M_{out}, v_{out} \models p_{out}$ *sse* $p_{out} \in L_{out}(v_{out})$.
2. $M_{out}, v_{out} \models \neg\alpha_{out}$ *sse* $M_{out}, v_{out} \not\models \alpha_{out}$.
3. $M_{out}, v_{out} \models \alpha_{out1} \wedge \alpha_{out2}$ *sse* $M_{out}, v_{out} \models \alpha_{out1}$ *e* $M_{out}, v_{out} \models \alpha_{out2}$.
4. $M_{out}, v_{out} \models \alpha_{out1} \vee \alpha_{out2}$ *sse* $M_{out}, v_{out} \models \alpha_{out1}$ *ou* $M_{out}, v_{out} \models \alpha_{out2}$.
5. $M_{out}, v_{out} \models \alpha_{out1} \rightarrow \alpha_{out2}$ *sse* $M_{out}, v_{out} \models \alpha_{out2}$ *sempre que* $M_{out}, v_{out} \models \alpha_{out1}$.
6. $M_{out}, v_{out} \models AX\alpha_{out}$ *sse para todo* v_{out_2} *tal que* $v_{out} \rightarrow_{out} v_{out_2}$, *temos que* $M_{out}, v_{out_2} \models \alpha_{out}$.
7. $M_{out}, v_{out} \models EX\alpha_{out}$ *sse existe algum* v_{out_2} *tal que* $v_{out} \rightarrow_{out} v_{out_2}$ *e* $M_{out}, v_{out_2} \models \alpha_{out}$.
8. $M_{out}, v_{out} \models AG\alpha_{out}$ *sse para todo caminho* $v_{out} = v_{out_1} \rightarrow_{out} v_{out_2} \rightarrow_{out} v_{out_3} \rightarrow_{out} \ldots$ *e para todo* v_{out_i} *para* $i \geq 1$ *ao longo do caminho, temos que* $M_{out}, v_{out_i} \models \alpha_{out}$.
9. $M_{out}, v_{out} \models EG\alpha_{out}$ *sse existe algum caminho* $v_{out} = v_{out_1} \rightarrow_{out} v_{out_2} \rightarrow_{out} v_{out_3} \rightarrow_{out} \ldots$ *e para todo* v_{out_i} *para* $i \geq 1$ *ao longo do caminho, temos que* $M_{out}, v_{out_i} \models \alpha_{out}$.
10. $M_{out}, v_{out} \models AF\alpha_{out}$ *sse para todo caminho* $v_{out} = v_{out_1} \rightarrow_{out} v_{out_2} \rightarrow_{out} v_{out_3} \rightarrow_{out} \ldots$ *existir algum* v_{out_i} *para* $i \geq 1$ *ao longo do caminho tal que* $M_{out}, v_{out_i} \models \alpha_{out}$.
11. $M_{out}, v_{out} \models EF\alpha_{out}$ *sse existe algum caminho* $v_{out} = v_{out_1} \rightarrow_{out} v_{out_2} \rightarrow_{out} v_{out_3} \rightarrow \ldots$ *tal que existe algum* v_{out_i} *para* $i \geq 1$ *ao longo deste caminho tal que* $M_{out}, v_{out_i} \models \alpha_{out}$.
12. $M_{out}, v_{out} \models AU(\alpha_{out_1}, \alpha_{out_2})$ *sse todos os caminhos* $v_{out} = v_{out_1} \rightarrow_{out} v_{out_2} \rightarrow_{out} v_{out_3} \rightarrow_{out}$ *... temos que existe algum* $i \geq 1$ *tal que* $M_{out}, v_{out_i} \models \alpha_{out_2}$ *e para todo j tal que* $1 \leq j \leq i - 1$, *temos que* $M_{out}, v_{out_j} \models \alpha_{out_1}$.
13. $M_{out}, v_{out} \models EU(\alpha_{out_1}, \alpha_{out_2})$ *sse existe um caminho* $v_{out} = v_{out_1} \rightarrow_{out} v_{out_2} \rightarrow_{out} v_{out_3} \rightarrow_{out}$ *... tal que que existe algum* $i \geq 1$ *tal que* $M_{out}, v_{out_i} \models \alpha_{out_2}$ *e para todo j tal que* $1 \leq j \leq i - 1$, *temos que* $M_{out}, v_{out_j} \models \alpha_{out_1}$.
14. $M_{out}, v_{out} \models i$ *sse* $N(i) = v_{out}$.
15. $M_{out}, v_{out} \models @_i\alpha_{out}$ *sse* $M_{out}, v \models \alpha_{out}$, *onde* $N(i) = v$.
16. $M_{out}, v_{out} \models IN\alpha_{in}$ *sse* $M_{in}, v_{0in} \models \alpha_{in}$, *onde* $C(v_{out}) = M_{in}$.
17. $M_{in}, v_{in} \models p_{in}$ *sse* $p_{in} \in L_{in}(v_{in})$.
18. $M_{in}, v_{in} \models \neg\alpha$ *sse* $M_{in}, v_{in} \not\models \alpha_{in}$.
19. $M_{in}, v_{in} \models \alpha_{in1} \wedge \alpha_{in2}$ *sse* $M_{in}, v_{in} \models \alpha_{in1}$ *e* $M_{in}, v_{in} \models \alpha_{in2}$.
20. $M_{in}, v_{in} \models \alpha_{in1} \vee \alpha_{in2}$ *sse* $M_{in}, v_{in} \models \alpha_{in1}$ *ou* $M_{in}, v_{in} \models \alpha_{in2}$.

21. $M_{in}, v_{in} \models \alpha_{in1} \rightarrow \alpha_{in2}$ *sse* $M_{in}, v_{in} \models \alpha_{in2}$ *sempre que* $M_{in}, v_{in} \models \alpha_{in1}$.
22. $M_{in}, v_{in} \models AX\alpha_{in}$ *sse para todo* v_{in_2} *tal que* $v_{in} \rightarrow_{in} v_{in_2}$, *temos que* $M_{in}, v_{in_2} \models \alpha_{in}$.
23. $M_{in}, v_{in} \models EX\alpha_{in}$ *sse existe algum* v_{in_2} *tal que* $v_{in} \rightarrow_{in} v_{in_2}$ *e* $M_{in}, v_{in_2} \models \alpha_{in}$.
24. $M_{in}, v_{in} \models AG\alpha_{in}$ *sse para todo caminho* $v_{in} = v_{in_1} \rightarrow_{in} v_{in_2} \rightarrow_{in} v_{in_3} \rightarrow_{in} ...$ *e para todo* v_{in_i} *para* $i \geq 1$ *ao longo do caminho, temos que* $M_{in}, v_{in_i} \models \alpha_{in}$.
25. $M_{in}, v_{in} \models EG\alpha_{in}$ *sse existe algum caminho* $v_{in} = v_{in_1} \rightarrow_{in} v_{in_2} \rightarrow_{in} v_{in_3} \rightarrow_{in} ...$ *e para todo* v_{in_i} *para* $i \geq 1$ *ao longo do caminho, temos que* $M_{in}, v_{in_i} \models \alpha_{in}$.
26. $M_{in}, v_{in} \models AF\alpha_{in}$ *sse para todo caminho* $v_{in} = v_{in_1} \rightarrow_{in} v_{in_2} \rightarrow_{in} v_{in_3} \rightarrow_{in} ...$ *existir algum* v_{in_i} *para* $i \geq 1$ *ao longo do caminho tal que* $M_{in}, v_{in_i} \models \alpha_{in}$.
27. $M_{in}, v_{in} \models EF\alpha_{in}$ *sse existe algum caminho* $v_{in} = v_{in_1} \rightarrow_{in} v_{in_2} \rightarrow_{in} v_{in_3} \rightarrow ...$ *tal que existe algum* v_{in_i} *para* $i \geq 1$ *ao longo deste caminho tal que* $M_{in}, v_{in_i} \models \alpha_{in}$.
28. $M_{in}, v_{in} \models AU(\alpha_{in_1}, \alpha_{in_2})$ *sse todos os caminhos* $v_{in} = v_{in_1} \rightarrow_{in} v_{in_2} \rightarrow_{in} v_{in_3} \rightarrow_{in} ...$ *temos que existe algum* $i \geq 1$ *tal que* $M_{in}, v_{in_i} \models \alpha_{in_2}$ *e para todo j tal que* $1 \leq j \leq i$ *- 1, temos que* $M_{in}, v_{in_j} \models \alpha_{in1}$
29. $M_{in}, v_{in} \models EU(\alpha_{in1}, \alpha_{in2})$ *sse existe um caminho* $v_{in} = v_{in_1} \rightarrow_{in} v_{in_2} \rightarrow_{in} v_{in_3} \rightarrow_{in} ...$ *e tal que existe algum* $i \geq 1$ *tal que* $M_{in}, v_{in_i} \models \alpha_{in_2}$ *e para todo j tal que* $1 \leq j \leq i$ *- 1, temos que* $M_{in}, v_{in_j} \models \alpha_{in1}$
30. $M_{in}, v_{in} \models AP(\alpha_{in})$ *sse para todo caminho* $v_{0_{in}} = v_{in_1} \rightarrow_{in} v_{in_2} \rightarrow_{in} v_{in_3} \rightarrow_{in} ... \rightarrow_{in} v_{in}$ *existir algum* v_{in_i} *para* $i \geq 1$ *ao longo do caminho tal que* $M_{in}, v_{in_i} \models \alpha_{in}$.
31. $M_{in}, v_{in} \models EP(\alpha_{in})$ *sse existe um caminho* $v_{0_{in}} = v_{in_1} \rightarrow_{in} v_{in_2} \rightarrow_{in} v_{in_3} \rightarrow_{in} ... \rightarrow_{in} v_{in}$ *e existe algum* v_{in_i} *para* $i \geq 1$ *ao longo do caminho tal que* $M_{in}, v_{in_i} \models \alpha_{in}$.

As cláusulas 14 e 15 são sobre os nominais e o conectivo $@_i$. Usamos isso para identificar e acessar uma versão específica. Logo, não precisamos de nominais e do operador $@_i$ para o nível do grafo de chamadas, apenas para o nível de grafo de versões. As cláusulas 30 e 31 são sobre conectivos temporais que falam sobre o passado. Usamos esses apenas para o grafo de chamadas, pois existem especificações que necessitam analisar chamadas de métodos que ocorreram no passado da execução do método atual, conforme veremos em um exemplo da próxima seção. Para o grafo de versões, não temos necessidade desses conectivos. As especificações avaliadas no grafo de versões analisam as versões do passado para o futuro, sem a necessidade de olhar para o passado. Caso essa necessidade surja, modificar a sintaxe e a semântica da lógica para incluir esses conectivos no nível de grafo de versões não é um problema.

O conectivo *IN*, cuja semântica está explicada na cláusula 16, é responsável pela comunicação entre os dois níveis de verificação com os quais trabalhamos. Esse conectivo é responsável por passar a verificação dos grafos de versões para os grafos de chamadas: observe na semântica que a fórmula no escopo do conectivo *IN* é avaliada no grafo de chamadas correspondente ao estado do grafo de versões. Esse grafo de chamadas é obtido por meio da função C. Quando a avaliação da fórmula no grafo de chamadas termina, a verificação volta para o grafo de versões para o mesmo estado de antes, já que o conectivo *IN* não interfere esse ponto. Dessa forma, lidamos com os dois níveis de verificação de maneira uniforme. Esse comportamento do conectivo *IN* fica mais claro nos exemplos apresentados na seção a seguir.

4.4 Exemplos

Assumimos que a verificação no grafo de versões começa sempre a partir do primeiro estado, que representa a primeira versão do software. Porém, podemos tornar isso explícito usando o operador @. Com esse operador, também podemos pular para versões específicas. No caso do grafo de chamadas, assumimos que a verificação é sempre a partir do estado inicial, conforme definimos na semântica da lógica proposta.

Seja M_{out} uma estrutura de Kripke que representa um grafo de versões, conforme definimos na Seção 4.2. Verificamos se $M_{out} \models \alpha_{out}$, ou seja, se $M_{out}, v_{0_{out}} \models \alpha_{out}$.

Exemplo 10 (Exemplos gerais de consultas) *Apresentamos neste exemplo algumas especificações de software adaptadas de (CZEPA et al., 2017). O esquema geral dessas consultas é apresentar uma fórmula que codifica a propriedade que deve ser analisada no grafo de chamadas, α_{in}, e então a fórmula α_{out}, que é avaliada no grafo de versões e utiliza a fórmula α_{in}. Fazemos assim por questões de organização e legibilidade das fórmulas.*

Especificação 01: *o método convertToIntervalXml da classe ConversionFilter é executado pelo software em todas as versões exceto na versão 5.*

- $\alpha_{in} := EF(ConversionFilter \wedge convertToInternalXml)$.

  Essa fórmula é verdadeira se encontramos um estado do grafo de chamadas que satisfaz $ConversionFilter \wedge convertToInternalXml$, de acordo com a semântica do conectivo EF. Isso significa que ocorre uma chamada ao método

convertToIntervalXml da classe ConversionFilter, ou seja, o método em questão é executado pelo software.

- $\alpha_{out} := AG((i_5 \rightarrow \neg IN(\alpha_{in})) \wedge (\neg i_5 \rightarrow IN(\alpha_{in})))$.

Verificamos para todos os estados do grafo de versões codificado como uma estrutura de Kripke se $(i_5 \rightarrow \neg IN(\alpha_{in})) \wedge (\neg i_5 \rightarrow IN(\alpha_{in}))$ *é verdadeira no estado. Usamos a implicação para verificar se estamos ou não na versão 5, por meio do nominal* i_5*. Se sim, então o grafo de chamadas dessa versão não deve satisfazer* α_{in}*. Para realizar essa verificação, usamos o conectivo IN. Caso não estejamos na versão 5, então o grafo de chamadas da versão deve sim satisfazer* α_{in}*. Logo, verificamos se o estado satisfaz* $IN(\alpha_{in})$.

Podemos usar o açúcar sintático Big And ($\bigwedge$*) e obter a seguinte fórmula:*

- $\alpha_{out} := @_{i_5}\neg IN(\alpha_{in}) \wedge \bigwedge_{i_j \in I}^{j \neq 5} @_{i_j} IN(\alpha_{in})$

O problema é que, dependendo do tamanho do grafo de versões, essa fórmula pode ser muito grande, já que teremos uma conjunção para cada vértice do grafo. Usando os operadores temporais, podemos construir fórmulas menores.

Especificação 02: *a partir da quinta versão, se ocorrer uma chamada ao método placeOrder da classe CustomerPortalServiceCLientProxy, então eventualmente deve acontecer uma chamada ao método forwardRequest da classe ApacheCamelBroker.*

- $\alpha_{in} := AG((CustomerPortalServiceClientProxy \wedge placeOrder) \rightarrow AF(ApacheCamelBroker \wedge forwardRequest))$.

A propriedade deve ser atendida por todos os estados do grafo de chamadas. Logo, usamos o conectivo AG. Como temos uma estrutura de se-então na especificação, usamos a implicação. Então, se tivermos uma chamada para o método placeOrder, o que é representado por encontrarmos um estado satisfazendo $CustomerPortalServiceClientProxy \wedge placeOrder$*, precisamos eventualmente ter uma chamada para o método forwardRequest. Isso é representado pelo conectivo* AF.

– $\alpha_{out} := @_{i_5} AG(IN\alpha_{in})$.

Como a propriedade deve ser satisfeita a partir da quinta versão, ignoramos as versões anteriores a ela e realizamos a verificação a partir dela. Para isso, usamos o conectivo $@_i$. Essa fórmula funciona por causa da característica do grafo de versões de possuir somente arestas para as próximas versões, sem arestas que tornem possível voltar para uma versão pela qual já passamos. As fórmulas propostas nesta seção levam essa característica em consideração. Como a propriedade codificada por α_{in} deve ser satisfeita por todos os estados a partir da quinta versão, usamos o conectivo AG. Para indicar que α_{in} deve ser verificada nos grafos de chamadas, usamos o conectivo IN.

– *Atente para a diferença entre as duas fórmulas:*
  – *$@_{i_5} AG(IN\alpha_{in})$: para todas as versões a partir da quinta, a fórmula α_{in} deve ser satisfeita pelos grafos de chamadas das versões. Em outras palavras, para todos os estados em todos os caminhos a partir do estado correspondente à versão 5 no grafo de versões, temos que os grafos de chamadas de cada versão satisfazem a fórmula α_{in}.*
  – *$@_{i_5} IN(AG\alpha_{in})$: o grafo de chamadas correspondente à versão 5 satisfaz a fórmula $AG\alpha_{in}$, ou seja, para todos os estados ao longo de todos os caminhos nesse grafo de chamadas, temos que a fórmula α_{in} é satisfeita.*

Especificação 03: *a classe CustomerDAO chama diretamente a classe CostumerObject até a versão 5.*

– $\alpha_{in} := EF(CustomerDAO \wedge EX(CostumerObject))$.

A fórmula no escopo do conectivo EF quer dizer que algum método da classe CustomerDAO chama algum método da classe CostumerObject. Logo, temos a chamada direta de classes. O conectivo EF é responsável por indicar que essa situação deve ocorrer em algum momento no grafo de chamadas.

– $\alpha_{out} := AU(IN\alpha_{in}, i_5)$.

Usamos o conectivo AU para verificar se α_{in} é satisfeita pelo grafo de

chamadas de todas as versões até a quinta versão. Atente para o uso do conectivo IN para informar que a verificação de α_{in} é com base no grafo de chamadas e não com base no grafo de versões.

Especificação 04: *para todas as versões do software deve ser válido que toda chamada do método generatePDF da classe DocumentGenerationFilter é precedida por uma chamada do método addAdditionalInformation da classe EnrichmentFilter.*

- $\alpha_{in} := AG((DocumentGenerationFilter \wedge generatePDF) \rightarrow AP(EnrichementFilter \wedge addAdditionalInformation))$.

Essa especificação segue a mesma estrutura da Especificação 02, porém, no lugar de procurarmos a ocorrência de uma chamada em algum momento do futuro, procuramos no passado por meio do conectivo AP.

- $\alpha_{out} := AG(IN(\alpha_{in}))$

Como a propriedade codificada por α_{in} deve ser verdadeira em todas as versões, usamos apenas o conectivo AG. Lembre-se que, conforme definimos no começo da seção, verificamos essas fórmulas a partir do estado inicial.

Exemplo 11 (Ilustrando o poder da semântica da lógica proposta) *Considere o seguinte exemplo que ilustra o poder da semântica que propomos:*

1. $\phi_1 := IN(\alpha_{in}) \wedge \alpha_{out}$.

   Seja M_{out} um grafo de versões e v_{out} um estado desse grafo de versões. Considere o que acontece quando $M_{out}, v_{out} \models \phi_1$:

   - *Pela semântica da conjunção, $v_{out} \models IN(\alpha_{in})$, o que significa que o grafo de chamadas associado ao estado v_{out} satisfaz α_{in}.*
   - *Pela semântica da conjunção, $v_{out} \models \alpha_{out}$, o que significa que o estado v_{out} satisfaz α_{out}.*
   - *Em resumo, a fórmula α_{in} deve ser satisfeita pelo grafo de chamadas, já a fórmula α_{out} deve ser satisfeita por um estado do grafo de versões.*

2. $\phi_2 := IN(\alpha_{in_1} \wedge \alpha_{in_2})$.

Seja M_{out} um grafo de versões e v_{out} um estado desse grafo de versões. Considere o que acontece quando $M_{out}, v_{out} \models \phi_2$:

- *Pela semântica do operador IN, devemos verificar se o grafo de chamadas associado ao estado v_{out} satisfaz as fórmulas α_{in_1} e α_{in_2}.*
- *Para ϕ_2, não verificamos se o estado v_{out} satisfaz α_{in_1} ou α_{in_2}, como fizemos para a fórmula α_{out} de ϕ_1. Por causa do escopo do operador IN, verificamos α_{in_1} e α_{in_2} com base no grafo de chamadas associado ao estado v_{out} pela função C.*

O que queremos enfatizar neste exemplo é o escopo do operador IN: com ele, podemos definir exatamente o que deve ser verificado em cima do grafo de chamadas e o que deve ser verificado em cima do grafo de versões. Com esse operador, podemos percorrer o grafo de versões entrando nos grafos de chamadas quando for necessário e então voltando para o grafo de versões para prosseguirmos com a verificação.

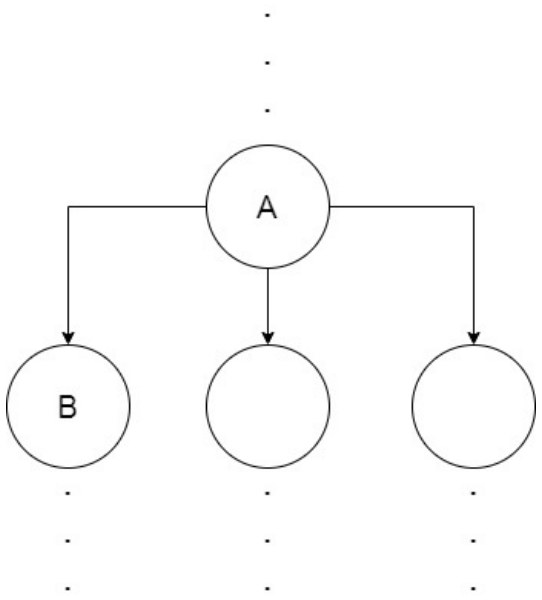
Exemplo 12 (Propriedades relacionadas à arquitetura de camadas) *Na Seção 2.1, apresentamos a arquitetura de camadas e mostramos algumas especificações relacionadas a esse modelo arquitetônico no exemplo 2.1. Agora, formalizamos essas especificações na lógica proposta. Essas especificações são a respeito dos grafos de chamadas.*

Nos anexos A, apresentamos as provas da corretude das fórmulas que apresentamos. Usamos átomos proposicionais com os mesmos nomes das classes e dos métodos.

1. ***A classe** A **chama diretamente a classe** B.*

Encontramos essa situação na Especificação 03 do Exemplo 4.4. Essa especificação é verdadeira se e somente se encontramos a seguinte situação no grafo de chamadas:

Figura 23 – Chamada direta entre classes



Fonte: elaborado pelo autor (2021).

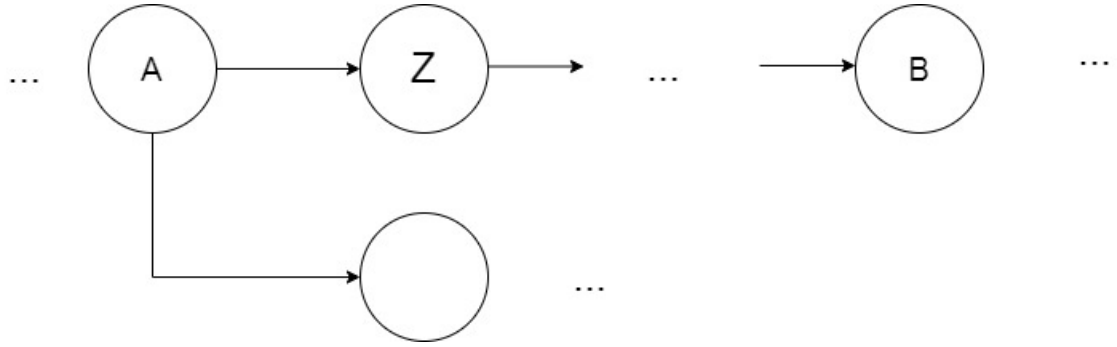
Com base nisso, escrevemos uma fórmula que é verdadeira se a situação apresentada acontece no grafo de chamadas:

EF(A ∧ EX(B)): em algum momento, algum método da classe A aparece na estrutura de Kripke e pelo menos um sucessor desse método é um método da classe B.

2. ***A classe** A **chama indiretamente a classe** B.*

Se sim, a seguinte situação ocorre no grafo de chamadas:

Figura 24 – Chamada indireta entre classes



Fonte: elaborado pelo autor (2021).

A seguinte fórmula é verdadeira se, e somente se, a situação descrita acontece no grafo de chamadas:

EF(A ∧ EX(¬B ∧ EF(B))): algum método da classe A aparece na estrutura de Kripke e pelo menos um sucessor desse método não é um método da classe B, porém existe uma sequência de chamadas de métodos que culmina em um método da classe B.

3. ***Existe uma sequência de chamadas partindo do método** m **que alcança o método** n.*

Procuramos na estrutura por uma situação análoga à chamada indireta entre classes. Porém, aqui não tem problema se o método m chamar diretamente o método n. Neste caso, significa que podemos ter uma sequência de chamadas de tamanho 0. Consideramos o tamanho das sequências como sendo o número de métodos chamados entre as chamadas de m e n. Logo, temos a fórmula:

EF(m ∧ EX EF(n))

Essa fórmula cobre tanto o caso de sequências de tamanho 0 como sequências de tamanhos arbitrários. Se quisermos evitar sequências de tamanho 0, usamos a fórmula EF(m ∧ EX EX EF(n)) para forçar que exista uma chamada ao método n em uma sequência de tamanho pelo menos 1. Atente que estamos considerando o tamanho da sequência de chamadas como sendo o número de estados, que representam os métodos, intermediários entre os estados correspondentes aos métodos m e n.

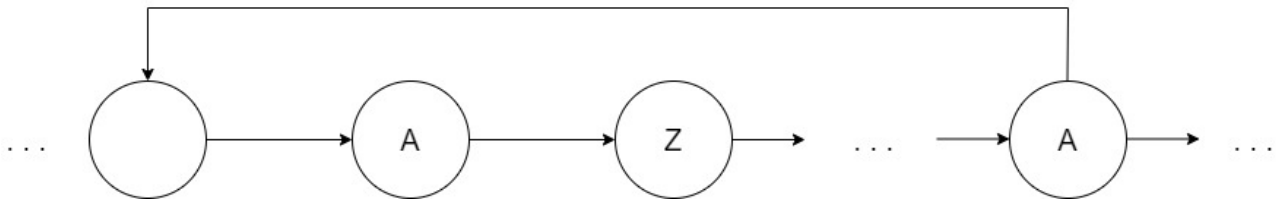
4. ***Existe uma sequência de chamadas partindo do método** m **que alcança o método** m.*

A situação que desejamos achar na estrutura de Kripke é como no caso anterior, mas usamos m no lugar de n. Note que, como temos apenas um vértice para cada método, em caso de resposta positiva para essa pergunta, temos que M está em um ciclo de chamadas. Temos a fórmula EF(m ∧ EX EF(m)) e, caso desejarmos evitar uma sequência de tamanho 0, usamos EF(m ∧ EX EX EF(m)), exatamente como fizemos anteriormente.

5. ***A classe A está em um laço de chamadas.***

Um método da classe A é chamado. Após essa chamada, algum método de A é sempre chamado novamente enquanto o programa continuar a ser executado. Procuramos no grafo de chamadas pela seguinte situação:

Figura 25 – Laço de chamadas



Fonte: elaborado pelo autor (2021).

Temos a fórmula:

$EF(A \wedge EX(EGEF(A)))$*: em algum momento, um método da classe A é chamado. Para pelo menos um sucessor desse método, temos um caminho no qual um método de A sempre volta a aparecer. Se isso ocorre, A está em um loop de chamadas.*

4.5 Algoritmo de Verificação de Modelos com os conectivos do passado

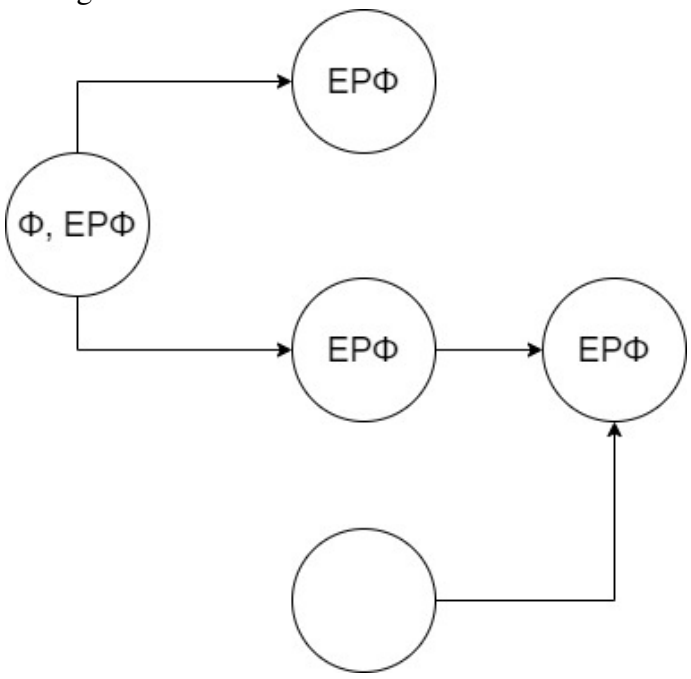
No Capítulo 3, explicamos um algoritmo de Verificação de Modelos para as lógicas CTL e híbrida básica. Esse algoritmo consiste em verificar se a estrutura de Kripke satisfaz a fórmula de entrada verificando quais estados satisfazem as suas subfórmulas, dos átomos proposicionais até a fórmula em si. Mostramos o que precisa ser feito para cada conectivo da CTL e da lógica híbrida básica. Esse mesmo algoritmo pode ser usado para a Verificação de Modelos da lógica que propomos neste capítulo: é suficiente explicarmos o que precisa ser calculado para os novos conectivos que incluímos, a saber, AP, EP, IN. Atente que o algoritmo que propomos aqui funciona em estruturas de Kripke arbitrárias, ou seja, não se trata de um algoritmo desenvolvido especificamente para grafos de chamadas ou grafos de versões. Logo, levamos em consideração estruturas de Kripke quaisquer ao explicar o algoritmo, como fizemos anteriormente no Capítulo 3.

4.5.1 Conectivo EP

Em uma estrutura de Kripke, um estado v satisfaz $EP\phi$ se possui um ancestral, i.e, um estado que aparece antes de v em um caminho, que satisfaz ϕ. Assim como o algoritmo funciona para os demais operadores temporais, precisamos rastrear os estados que atendem a essa condição e marcá-los com $EP\phi$. Contudo, o sentido da busca no grafo é invertido em relação ao algoritmo para os conectivos que falam sobre o futuro.

Para o algoritmo da CTL apresentado no capítulo anterior, analisamos os sucessores de um estado e com base nas fórmulas satisfeitas por eles, decidimos se esse estado satisfaz ou não à fórmula. Por exemplo, considere a Figura 17, que ilustra a maneira como decidimos se um estado satisfaz uma fórmula com o conectivo AF. Precisamos percorrer o grafo de trás para frente. Claramente, isso não é problema algum: podemos percorrer o grafo a partir de um vértice v seguindo o sentido das arestas que saem de v ou podemos seguir o sentido das arestas que entram em v. Esse último é o sentido ao qual nos referimos quando falamos em percorrer o grafo de trás para frente. Para os conectivos que falam sobre o passado, invertemos o sentido dessa busca e seguimos o sentido usual, que é a partir das arestas que saem do vértice. Observe:

Figura 26 – Funcionamento do conectivo *EP*



Fonte: elaborado pelo autor (2021).

Os estados marcados com $EP\phi$ possuem um estado v marcado com ϕ como ancestral. Precisamos percorrer os caminhos que começam em v marcando os estados com $EP\phi$. Logo, em

um primeiro momento, marcamos os estados que satisfazem ϕ e depois, a partir deles, marcamos os estados que satisfazem $EP\phi$.

Como ϕ é uma fórmula menor, i.e, com menos conectivos, do que $EP\phi$, os estados que foram marcados com ϕ já foram calculados em um passo anterior do algoritmo de marcação. Logo, o procedimento é o seguinte:

1. Marcamos com $EP\phi$ todos os estados que foram marcados com ϕ. Seja W esse conjunto de estados.
2. Seguimos os caminhos que começam nos estados de W marcando com $EP\phi$ os estados que estão nesses caminhos.

Para demonstrar a corretude desse procedimento, procedemos como segue. Definimos que $[\phi]$ é o conjunto de estados que satisfazem a fórmula ϕ. Definimos:

Definição 4.5.1 (Conjuntos *next*) *Seja V o conjunto de estados de uma estrutura de Kripke e considere as seguintes definições:*

1. $next_{\exists}(X) = \{v' \in V \mid$ *existe v tal que* $v \rightarrow v'$ *e* $v \in X\}$
2. $next_{\forall}(X) = \{v' \in V \mid$ *para todo v tal que* $v \rightarrow v'$*, temos que* $v \in X\}$

Definição 4.5.2 (Função monotônica) *Seja V um conjunto de estados e* $F : 2^V \rightarrow 2^V$ *uma função.*

1. *F é monotônica se e somente se* $X \subseteq Y$ *implica que* $F(X) \subseteq F(Y)$ *para todos os subconjuntos X e Y de V.*
2. *Um subconjunto X de V é chamado de ponto fixo se e somente se* $F(X) = X$.

Agora, considere o seguinte teorema, que pode ser encontrado em (RYAN; HUTH, 2004):

Teorema 4.5.1 (Pontos fixos de funções monotônicas) *Seja V um conjunto com n + 1 elementos. Se* $F : 2^V \rightarrow 2^V$ *é uma função monotônica, então:*

1. $F^{n+1}(\emptyset)$ *é o menor ponto fixo de F.*
2. $F^{n+1}(V)$ *é o maior ponto fixo de F.*

Considere a seguinte função $F : 2^V \rightarrow 2^V$. Provamos que F é monotônica logo em seguida.

$$F(X) = [\phi] \cup next_{\exists}(X)$$

Prova: conforme a definição de função monotônica, precisamos mostrar que se $X \subseteq Y$, então $F(X) \subseteq F(Y)$, onde X e Y são subconjuntos quaisquer de V.

O conjunto $[\phi]$ é fixo. Estamos realizando a união de conjuntos. Então podemos apenas adicionar elementos novos e não remover elementos. Sendo assim, é suficiente provarmos que $next_{\exists}(X) \subseteq next_{\exists}(Y)$, se $X \subseteq Y$.

Seja $v \in X$. Logo, $v \in Y$, pois estamos assumindo que $X \subseteq Y$. Se existe v' tal que $v \rightarrow v'$, $v' \in next_{\exists}(X)$ de acordo com a definição do conjunto $next_{\exists}(X)$. Como $v \in Y$, a existência de um tal v' implica que $v' \in next_{\exists}(Y)$, novamente pela definição do conjunto $next_{\exists}(Y)$. Logo, $next_{\exists}(X) \subseteq next_{\exists}(Y)$. Portanto, F é monotônica.

C.Q.D

Pelo teorema que apresentamos acima, $F^{n+1}(\emptyset)$, onde n é o número de elementos no conjunto V no qual a função está definida, é o menor ponto fixo de F. Vamos mostrar que $[EP\phi]$ = $F^{n+1}(\emptyset)$:

Teorema 4.5.2 (Corretude do conectivo EP) *Sejam $M = (V, \rightarrow, L, PROP)$ uma estrutura de Kripke, $[EP\phi]$ o conjunto de estados de M que satisfazem a fórmula $EP\phi$, n o número de estados em V e $F : 2^V \rightarrow 2^V$ uma função definida como $F(X) = [\phi] \cup next_{\exists}(X)$. Temos que $[EP\phi] = F^{n+1}(\emptyset)$.*

Prova: temos uma igualdade entre conjuntos. Logo, temos que provar que $v \in [EP\phi]$ se e somente se $v \in F^{n+1}(\emptyset)$, onde $v \in V$.

$v \in [EP\phi]$ se e somente se $v \models \phi$ ou v está em um caminho tal que $v' \models \phi$ para algum ancestral v' de v, ou seja, v está em um caminho que parte de um estado v' que satisfaz ϕ.

Analisando os estados que estão em $F^{n+1}(\emptyset)$, temos que:

– $F(\emptyset) = [\phi]$. Logo, temos os estados que satisfazem ϕ.

- $F^2(\emptyset) = [\phi] \cup next_{\exists}([\phi])$. Logo, temos os estados que satisfazem ϕ e os seus sucessores.
- A cada iteração i, adicionamos os sucessores dos estados que estão em $F^{i-1}(\emptyset)$.
- Se continuarmos assim, em $n+1$, como temos a garantia de ponto fixo pelo Teorema 4.5.1.3**??**, teremos todos os estados que satisfazem ϕ e todos os estados que estão em caminhos que partem dos estados que satisfazem ϕ.

Dessa análise, notamos que ambos os conjuntos possuem o mesmo tipo de estados. Logo, $[EP\phi] = F^{n+1}(\emptyset)$.

C.Q.D

Para a segunda parte da prova acima, ao demonstrar quais estados estão em $F^{n+1}(\emptyset)$, devemos usar indução matemática. Omitimos essa parte da prova aqui, porém a incluímos nos Apêndice B.

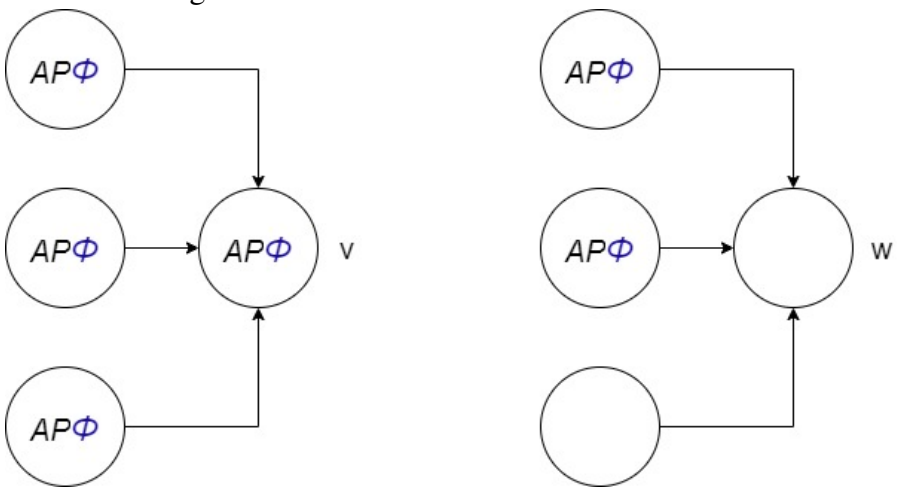
O procedimento para computar $[EP\phi]$ é como segue. Temos que SAT(ϕ) é uma função que computa quais estados satisfazem ϕ. Esse procedimento é apenas uma implementação da função F que apresentamos anteriormente. As iterações ocorrem até atingirmos um ponto fixo: como F é monotônica, o Teorema 4.5.1 garante a existência desse ponto fixo em até $n+1$ iterações. Logo, temos certeza que o procedimento termina e está correto, conforme o Teorema 4.5.1.

1. X $\leftarrow$ SAT(ϕ)
2. Y $\leftarrow \emptyset$
3. while X $\neq$ Y
4. $\quad$ Y $\leftarrow$ X
5. $\quad$ X $\leftarrow$ X $\cup$ $next_{\exists}(X)$
6. return X

4.5.2 Conectivo AP

A configuração que buscamos encontrar na estrutura de Kripke para concluir que um estado satisfaz a fórmula $AP\phi$ é mostrada no lado esquerdo da imagem a seguir:

Figura 27 – Funcionamento do conectivo AP



Fonte: elaborado pelo autor (2021).

Podemos concluir que o estado v satisfaz $AP\phi$, pois todos os seus antecessores satisfazem $AP\phi$. Já o estado w, como temos um antecessor desse estado que não satisfaz $AP\phi$, $w \not\models AP\phi$. O procedimento para computar os estados que satisfazem $AP\phi$ é análogo ao cálculo para $EP\phi$, porém todos os antecessores de um estado devem satisfazer $AP\phi$ e não apenas um.

1. Marcamos com $AP\phi$ todos os estados que foram marcados com ϕ em um passo anterior do algoritmo de marcação. Seja W esse conjunto de estados.
2. A partir dos estados de W, prosseguimos para os estados seguintes no caminho verificando se todos os antecessores desses estados estão marcados com $AP\phi$. Se sim, marcamos com $AP\phi$.

Para a corretude desse procedimento, prosseguimos como na subseção anterior. Considere a função: $G(X) = [\phi] \cup next_{\forall}(X)$

Para demonstrar que G é uma função monotônica, procedemos como no caso do conectivo EP:

Prova: é suficiente mostrar que se $X \subseteq Y$, então $next_{\forall}(X) \subseteq next_{\forall}(Y)$, pelo mesmo motivo do caso anterior, para o conectivo EP.

Se $v' \in next_{\forall}(X)$, então para todo v tal que $v \to v'$, temos que $v \in X$, conforme a definição de $next_{\forall}(X)$. Consequentemente, $v \in Y$, pois $X \subseteq Y$. Logo, $v' \in next_{\forall}(Y)$, pois todos os estados v tais que $v \to v'$ estão em Y. Portanto, $next_{\forall}(X) \subseteq next_{\forall}(Y)$. Consequentemente, G é uma função monotônica.

C.Q.D

Assim como para o conectivo EP, temos que $[AP\phi] = G^{n+1}(\emptyset)$:

Teorema 4.5.3 (Corretude do conectivo AP) *Sejam $M = (V, \rightarrow, L, PROP)$ uma estrutura de Kripke, $[AP\phi]$ o conjunto de estados de M que satisfazem a fórmula $AP\phi$, n o número de estados em V e $G : 2^V \rightarrow 2^V$ uma função definida como $G(X) = [\phi] \cup next_\forall(X)$. Temos que $[AP\phi] = G^{n+1}(\emptyset)$.*

A prova é análoga à prova do caso anterior: trocamos $next_\exists(X)$ por $next_\forall(X)$.

O procedimento também é análogo ao caso anterior, inclusive em relação à terminação:

1. X $\leftarrow$ SAT(ϕ)
2. Y $\leftarrow \emptyset$
3. while X $\neq$ Y
4. Y $\leftarrow$ X
5. X $\leftarrow$ X $\cup$ $next_\forall(X)$
6. return X

4.6 Complexidade

Nesta seção, explicamos qual é a complexidade do algoritmo de Verificação de Modelos para a lógica que propomos neste capítulo, com base na estrutura que propomos na Seção 4.2. Analisamos a complexidade dos novos conectivos que a lógica proposta (Definição 4.3) introduz em relação às duas lógicas que apresentamos no Capítulo 3, primeiramente para o nível 1 e em seguida, para o nível 2.

Lembre-se da complexidade da Verificação de Modelos para a CTL, sem os conectivos para falar sobre o passado, que apresentamos no Capítulo 3:

1. n é o número de estados da estrutura de Kripke.
2. m é o número de transições na estrutura de Kripke.
3. t é tamanho da fórmula, i.e, o número de conectivos na fórmula.
4. A complexidade da Verificação de Modelos para a CTL é $O(t \times n \times (n+m))$.

Para os conectivos EP e AP, mudamos apenas a direção na qual percorremos a estrutura de Kripke em relação aos conectivos que falam sobre o futuro. Portanto, a complexidade da Verificação de Modelos com os conectivos do passado, correspondente ao nível 1 de verificação, continua a mesma que apresentamos acima (quarto item) e é polinomial na entrada do problema.

Para o nível 2 de verificação, temos uma fórmula α_{out} da lógica proposta neste capítulo e um grafo de versões M_{out} (de fato, uma estrutura de Kripke que representa um grafo de versões, porém vamos chamar apenas de grafo de versões, conforme falamos anteriormente) e queremos calcular o custo da Verificação de Modelos dessa fórmula nesse grafo. Em α_{out}, temos conectivos $@, IN$, temporais e booleanos. Para os conectivos temporais e booleanos, o algoritmo de Verificação de Modelos funciona como na CTL apresentada no Capítulo 3, já que são os mesmos conectivos avaliados em uma estrutura de Kripke. Logo, a complexidade para esses conectivos é a mesma. Precisamos avaliar o impacto de verificar os conectivos $@, IN$. Para os átomos proposicionais e nominais, já vimos nas seções 3.4 e 3.6 que o custo de computar a verificação deles é dominado pelos demais custos, por isso não os incluímos nesta análise. Considere:

1. n_{out} é o número de estados do grafo de versões M_{out}.
2. m_{out} é o número de transições do grafo de versões M_{out}.
3. t_{out} é o número de conectivos booleanos e temporais na fórmula α_{out} avaliada no grafo de versões M_{out}.

Desconsiderando os conectivos IN e $@$, consequentemente, levando em consideração apenas os conectivos booleanos e temporais, o custo da Verificação de Modelos para a fórmula que é verificada no grafo de versões é a complexidade da CTL que já discutimos no Capítulo 3 e que relembramos acima, ou seja, $O(t_{out} \times n_{out} \times (n_{out} + m_{out}))$. Seja $v_1 = t_{out} \times n_{out} \times (n_{out} + m_{out})$.

O custo v_1 é um custo que sempre pagamos ao realizar a verificação. Analisando o custo dos conectivos $@$ e IN, segue que:

1. O conectivo $@$ nos faz percorrer o grafo de versões procurando uma determinada versão. Logo, sendo y o número de conectivos $@$ na fórmula, a complexidade é $O(y \times (n_{out} + m_{out})$. Essa complexidade é dominada por v_1, pois espera-se que y seja um valor consideravelmente menor do que $t_{out} \times n_{out}$: y é o número de apenas um conectivo, enquanto t_{out} contempla

conectivos temporais e booleanos, além de n_{out} ser o número de vértices no grafo de versões. (Lembre-se que $(n_{out} + m_{out})$ é o custo de percorrer um grafo.)

2. Para o conectivo *IN*:
   - Seja z o número de conectivos *IN* na fórmula α_{out} a ser verificada.
   - Para cada conectivo *IN*, precisamos avaliar uma fórmula no grafo de chamadas da versão correspondente.
   - Temos um conjunto de grafos de chamadas $GC = \{G_1, G_2, ..., G_{n_{out}}\}$ e um conjunto de fórmulas $A = \{\alpha_{in_1}, \alpha_{in_2}, ..., \alpha_{in_z}\}$ que devem ser avaliadas em um ou mais desses grafos, presentes no escopo dos conectivos *IN* existentes na fórmula α_{out}.
   - No pior caso, temos que avaliar cada uma das fórmulas em A em cada um dos grafos de chamadas em GC. Um exemplo no qual isso acontece é para uma fórmula do tipo $AG(IN(\beta_{out}))$.
   - Estabelecemos um limite superior para o custo apontado acima. Considere a maior fórmula em A e o maior grafo de chamadas (de acordo com o número de vértices e arestas) em GC. Considere $c_{max_{in}}$ como sendo o custo de computar a Verificação de Modelos dessa maior fórmula em cima desse maior grafo de chamadas.
   - Temos que $c_{max_{in}} = t_{max_{in}} \times n_{max_{in}} \times (n_{max_{in}} + m_{max_{in}})$, que corresponde ao custo da Verificação de Modelos para a CTL com conectivos do passado, conforme explicamos no terceiro parágrafo desta seção.
   - $t_{max_{in}}$ é o tamanho da maior fórmula em A, $n_{max_{in}}$ é o número de estados do maior grafo de chamadas em GC, $m_{max_{in}}$ é o número de transições desse grafo.
   - Logo, temos o custo $O(n_{out} \times z \times c_{max_{in}})$, pois no pior caso temos que pagar $c_{max_{in}}$ para cada estado do grafo de versões e para cada conectivo *IN* na fórmula α_{out}.
   - Seja $v_2 = n_{out} \times z \times c_{max_{in}}$.

Avaliamos todas as fórmulas no escopo de conectivos *IN*, o que tem custo v_2, e então resolvemos a Verificação de Modelos para a fórmula em cima do grafo de versões, o que tem custo v_1. Uma vez que já sabemos os valores-verdade das fórmulas no escopo dos conectivos *IN*, essas são tratadas como se fossem átomos proposicionais. Logo, a complexidade é $O(v_1 + v_2)$, que é polinomial na entrada. Consequentemente, o problema de Verificação de Modelos para a lógica proposta está na classe de complexidade P.

Comparando a complexidade do problema de Verificação de Modelos de outras

lógicas, aplicado à estrutura que propomos na Seção 4.2, obtermos a seguinte tabela, onde abreviamos Lógica Híbrida Básica para LHB:

Problema	Classe de complexidade	Referências
Verificação de Modelos da LTL	PSPACE	1
Verificação de Modelos da CTL	P	2
Verificação de Modelos da LHB	P	Veja a Seção 3.6
Verificação da lógica proposta	P	Calculada nesta seção

1. (VARDI; WILKE, 2008)
2. (Beyersdorff *et al.*, 2009; CLARKE *et al.*, 1986)

Para calcular essas complexidades, procedemos da mesma maneira que calculamos a complexidade da Verificação de Modelos para a lógica proposta: aplicamos o algoritmo nos grafos de chamadas (nível 1) para as fórmulas que estão no escopo de conectivos *IN* e, em seguida, podemos aplicar o algoritmo no grafo de versões (nível 2) tratando as fórmulas *IN* como átomos proposicionais. Para a CTL, o cálculo fica como o da lógica proposta nesta seção, já que os custos relacionados aos conectivos das lógicas híbridas foram dominados pelos custos relacionados aos conectivos temporais. Para a LHB, o cálculo é análogo, porém ficamos apenas com os custos de percorrer os grafos. Logo, a complexidade também é polinomial. Para a LTL, novamente, o cálculo é análogo. Como o problema em uma simples estrutura de Kripke já está em PSPACE completo, com a estrutura que propomos aqui, não é possível que a complexidade esteja em uma classe inferior. Como os custos da Verificação de Modelos no nível 1 e no nível 2 são somados e não multiplicados, não é possível que a complexidade passe para uma classe de complexidade superior. Logo, o problema de Verificação de Modelos da LTL com a estrutura descrita na Seção 4.2 está em PSPACE, assim como para uma simples estrutura de Kripke, como na definição 3.2. Um algoritmo de Verificação de Modelos para a LTL pode ser encontrado em (RYAN; HUTH, 2004; VARDI; WILKE, 2008).

4.7 Avaliação e comparação com outros trabalhos

Nesta seção, avaliamos a metodologia que propomos de acordo com os seis critérios dos quais falamos sobre na Subseção 2.3.6 e depois atualizamos a tabela comparativa entre os diferentes métodos para auxiliar na Análise de Conformidade Arquitetural com o método proposto neste trabalho. Listamos a seguir os seis critérios avaliativos e como a metodologia

proposta se enquadra neles.

1. **Flexibilidade da linguagem de modelagem:** no nível 1, usamos como representação do sistema os grafos de chamadas. Usando uma lógica temporal, conseguimos capturar os aspectos dinâmicos da execução do sistema no que diz respeito às relações de chamadas e métodos e verificar especificações. Entretanto, para avaliar especificações sobre outros pontos do sistema, podemos trocar os grafos de chamadas por outra estrutura. Desde que essa outra estrutura possa ser representada como estruturas de Kripke, o que facilmente acontece devido à generalidade das estruturas de Kripke, nossa metodologia funciona conforme descrevemos aqui. A linguagem que usamos consegue capturar uma ampla gama de especificações, usando os átomos proposicionais para auxiliar no processo. Logo, assumimos que o método proposto por este trabalho cumpre esse critério.
2. **Suporte à modelagem arquitetural**: a metodologia que propomos precisa utilizar a arquitetura modelada para verificar se o sistema está de fato implementando-a. Não fornecemos maneiras de modelar a arquitetura. Logo, nossa metodologia não cumpre esse critério. Entretanto, isso não é um problema: conforme podemos ver na Seção 2.3, é normal que a metodologia foque apenas na verificação de propriedades relacionadas à arquitetura e deixe a tarefa de modelagem arquitetural para outra ferramenta.
3. **Suporte para integração de documentação de arquitetura**: ao formalizar uma especificação de *software* usando uma fórmula da lógica temporal proposta, estamos apresentando uma nova maneira de compreender e avaliar essa especificação. Além disso, as fórmulas operam nos grafos de chamadas, que descrevem um aspecto do *software*, o que contribui para o entendimento do funcionamento do sistema, e nos grafos de versões, o que facilita o entendimento do processo global de desenvolvimento do sistema, ajudando no gerenciamento do processo de desenvolvimento. A parte de modelagem arquitetural e verificação são mantidas independentes. O método proposto neste trabalho utiliza a documentação da arquitetura ao realizar a verificação e, conforme explanamos nas linhas anteriores, fornece novas visões a respeito dessa arquitetura, o que pode ser bastante útil para os engenheiros de *software*, especialmente para novos engenheiros que entram em um projeto e precisam compreender como o *software* funciona para modificá-lo e mantê-lo. Logo, nossa metodologia cumpre esse critério.
4. **Compreensibilidade**: utilizar lógicas temporais para formalizar especificações exige experi-

ência. Apesar das lógicas temporais serem mais simples do que lógicas de primeira ordem, ainda é necessário prática e, consequentemente, tempo, para dominar os procedimentos de especificação e avaliação de fórmulas. Como normalmente acontece ao se aprender uma nova linguagem, é mais fácil ler uma fórmula do que escrever uma fórmula representando uma especificação. Mesmo assim, compreender exatamente o que uma fórmula expressa requer um tempo e um esforço de aprendizagem maior do que em outras metodologias mais intuitivas e com muitos elementos gráficos. Por exemplo, considere os modelos de reflexão. Tenha em mente que nem todas as formações de engenheiros de *software* incluem o estudo de Linguagens Formais e Teoria da Computação, o que torna ainda mais complicado o aprendizado do uso de lógicas. Como a lógica que propomos é mais simples do que lógica de primeira ordem e os operadores temporais operam em grafos (estruturas de Kripke são grafos), bastante conhecidos e utilizados em diversas áreas, o que contribui para a compreensão da metodologia proposta, assumimos que o método proposto neste trabalho atende parcialmente ao critério em questão.

5. **Suporte à verificação em dois níveis**: essa é uma inovação que propomos neste trabalho. Conforme explicamos ao longo do texto, desenvolvemos um método de verificação de propriedades que trabalha em dois níveis e incluímos maneiras de lidar com esses dois níveis de maneira homogênea, conforme explicamos neste capítulo. Obviamente, a metodologia atende completamente a esse critério.
6. **Manutenção do histórico de versões**: os grafos de versões fazem isso. Em paralelo, sistemas de controle de versão, indispensáveis no desenvolvimento de *software* contemporâneo, contribuem para essa manutenção. Logo, a metodologia proposta neste trabalho atende completamente ao critério em questão.

Seguindo a proposta de (SCHRÖDER, 2020), temos três níveis de satisfatibilidade para cada critério, conforme apresentamos na Subseção 2.3.6. Repetimos esses critérios aqui por conveniência:

+: requisitos do critério atendidos;
o: requisitos do critério parcialmente atendidos;
-: requisitos do critério não atendidos.

Apresentamos novamente a tabela comparativa das metodologias de acordo com os critérios elencados, incluindo nela a metodologia deste trabalho. Claramente, o nosso método se encaixa nas abordagens baseadas em lógica.

Critério	1	2	3	4	5	6
Modelos de reflexão						
Sonorgraph	-	+	-	+	-	o
SAVE	-	+	-	+	-	o
Structure101	-	+	-	+	-	o
Teamscale	-	+	-	+	-	o
Baseadas em regras						
DCL	-	-	+	+	-	o
TamDera	o	-	+	o	-	o
Dicto	+	-	+	+	-	o
Macker	o	-	+	o	-	o
Lattix Architect	-	+	-	+	-	o
StyleBasedChecker	+	-	+	o	-	o
ArCatch	+	-	+	o	-	o
Baseadas em lógica						
Herold	+	+	+	-	-	o
SCL	-	-	+	-	-	o
LogEn	o	-	+	-	-	o
Malicious Detection	-	-	+	o	-	o
Metodologia Proposta	+	-	+	o	+	+
Baseadas em consultas						
CQL	+	-	+	o	-	o
.QL	+	-	+	o	-	o
jQAssistant	+	-	+	o	-	o
Especificações embutidas						
ArchFace	-	-	-	o	-	o
ArchJava	-	-	-	o	-	o

Quando comparamos o método que propomos aqui com os demais métodos baseados em lógica, temos:

1. Em relação aos critérios 1, 2, 3 e 4, apenas a metodologia de Herold supera a nossa no quesito número de critérios satisfeitos. Entretanto, essa metodologia utiliza lógica de primeira ordem, que é mais complicada do que a lógica temporal que propomos, e suas estruturas também são mais complicadas de se entender e manipular do que grafos de chamadas e versões.
2. A metodologia de Herold consegue atender ao critério 2, porém faz isso utilizando diversas estruturas de Primeira Ordem, inserindo umas dentro das outras por meio de relações. O resultado é pouco intuitivo e facilmente leva à confusão do usuário. Como o modelo arquitetural precisa ser bem compreendido para que possa ser devidamente implementado, essas características não são desejáveis. Além disso, é comum que uma visão geral da arquitetura do *software* seja repassada para *stakeholders* que pouco ou nada entendem de estruturas de primeira ordem e precisam de representações gráficas para compreender o modelo arquitetural. Logo, a forma como Herold modela a arquitetura não é apropriada para essa finalidade.

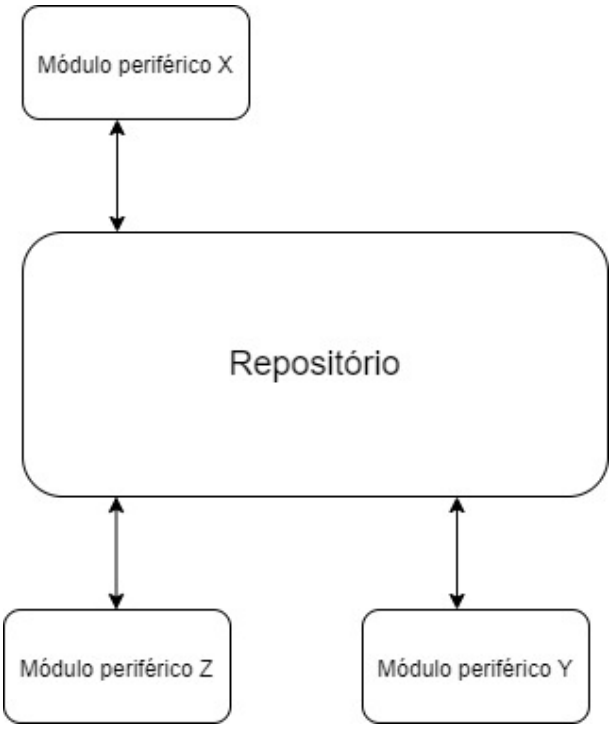
Quando comparamos com os métodos de outras abordagens, observamos que:

1. As abordagens baseadas em modelos de reflexão satisfazem ao critério 2. Devido à maneira como essa abordagem funciona, é necessário modelar a arquitetura com as mesmas estruturas e linguagem utilizadas durante o processo de verificação. Do contrário, não seria possível comparar os modelos. Em contrapartida, essas abordagens não atendem aos critérios 1 e 3. Nossa abordagem leva vantagem em relação ao critério 1, porém, em relação ao 3, precisamos ser cuidadosos ao realizar a comparação. Como o método de modelos de reflexão já modela a arquitetura em si, talvez não seja necessário fornecer o suporte para a integração de documentação. Logo, decidir se não satisfazer o critério 3 é realmente uma desvantagem requer uma análise caso a caso.
2. Nosso método não atende ao critério 2, porém isso é caso comum entre os métodos em geral. O foco maior desses métodos é verificar se a arquitetura pretendida está de fato sendo implementada. Logo, não possuir uma maneira de documentá-la não é necessariamente uma desvantagem. Podemos usar outras ferramentas para isso e apenas integrá-la ao método de verificação. Portanto, esse critério não ser atendido não é uma falha grave. Seria muito mais prejudicial não satisfazer a algum dos demais critérios.

5 ESTUDO DE CASO

Neste capítulo, analisamos com mais atenção um exemplo de arquitetura e especificações relacionadas a ele. Escolhemos a arquitetura de repositório, descrita no Exemplo 1.2.4. Esse padrão consiste em um repositório central acessado por módulos periféricos, conforme podemos ver na Figura 4. Os módulos periféricos não trocam informações entre si: as informações são trocadas apenas com o repositório. Logo, não podemos ter um método de uma classe pertencente a um módulo periférico X chamando um método de uma classe pertencente a um módulo periférico Y. Uma representação é apresentada abaixo:

Figura 28 – Modelo de um repositório genérico



Fonte: elaborado pelo autor (2021).

Usamos grafos de chamadas e grafos de versões para nos referirmos às estruturas de Kripke que codificam essas estruturas, conforme explicamos no capítulo anterior. Sendo assim, também nos referimos aos estados dessas estruturas chamando-os de vértices.

Considere um método A de uma classe C1 pertencente ao módulo X e um método B de uma classe C2 pertencente ao módulo Y. O método A não pode chamar o método B e vice-versa. A seguinte especificação expressa isso:

$$\alpha_{in1} := \neg(EF((A \wedge C1) \wedge EF(B \wedge C2))) \wedge \neg(EF((B \wedge C2) \wedge EF(A \wedge C1)))$$

Essa fórmula indica que não podemos encontrar uma execução do método A eventualmente seguida por uma execução do método B. A segunda fórmula expressa a mesma especificação, porém invertendo os papéis dos métodos A e B.

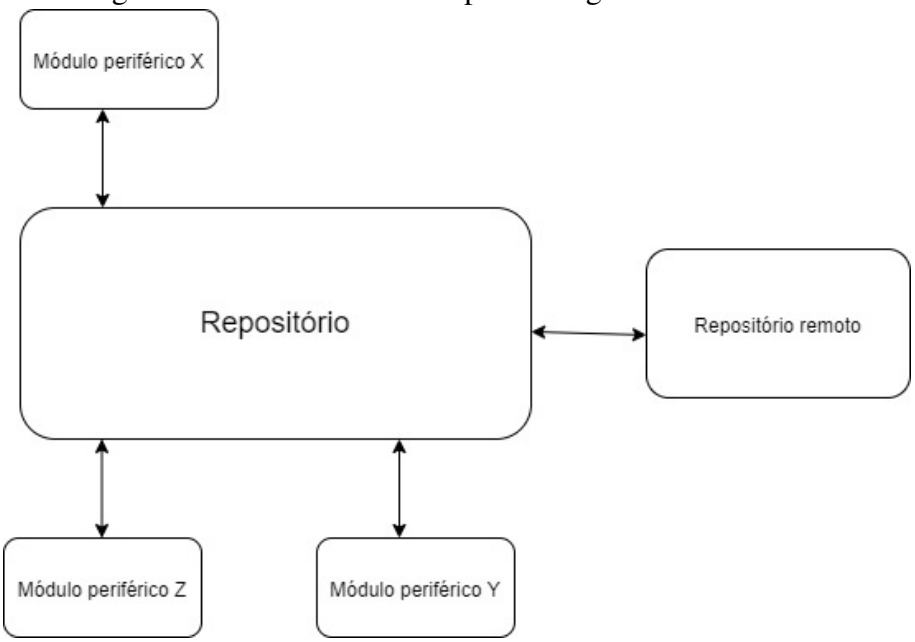
Como essa propriedade deve ser válida sempre, precisamos que essa fórmula seja válida em todas as versões. Isso é codificado pela fórmula a seguir:

$$\alpha_{out1} := AG(IN(\alpha_{in1}))$$

Logo, a propriedade codificada por α_{in1} é verificada em todos os vértices do grafo de versões ao aplicarmos a Verificação de Modelos.

Suponha um cenário no qual o repositório central inicialmente é apenas local, armazenado em algum servidor de uma universidade, por exemplo. A partir da quinta versão de um sistema que acessa esse repositório, uma cópia do repositório local deve ser armazenada em um servidor remoto por questões de segurança. Apresentamos o esquema abaixo:

Figura 29 – Modelo de um repositório genérico remoto



Fonte: elaborado pelo autor (2021).

Podemos assumir que existe um método, vamos chamá-lo de *performBackup*, que acessa o servidor remoto e o atualiza com os dados do repositório local. Esse método deve ser executado sempre que alguma alteração for realizada no repositório. Para simplificar, assumimos que existem três métodos que alteram o repositório: *updateData, createData, deleteData*.

Sempre que uma alteração for realizada no repositório local, essa alteração deve ser enviada para o repositório remoto. Logo, após a execução de um dos métodos que alteram o repositório ser realizada, o *backup* deve ser realizado. Temos as fórmulas a seguir:

$$\alpha_{in2} := AG(updateData \rightarrow EX(performBackup))$$
$$\alpha_{in3} := AG(createData \rightarrow EX(performBackup))$$
$$\alpha_{in4} := AG(deleteData \rightarrow EX(performBackup))$$

As fórmulas acima nos dizem que para todos os vértices do grafo de chamadas, se tivermos uma operação que altera as informações no repositório, logo em seguida devemos chamar o método que atualiza o servidor remoto. Essas fórmulas devem ser satisfeitas a partir da quinta versão, conforme dissemos anteriormente. Temos a seguinte fórmula a ser verificada no grafo de versões:

$$\alpha_{out2} := @_5 AG(IN(\alpha_{in2} \wedge \alpha_{in3} \wedge \alpha_{in4}))$$

A fórmula acima nos diz que, a partir da quinta versão, as fórmulas $\alpha_{in2}, \alpha_{in3}, \alpha_{in4}$ devem ser verdadeiras para todos os grafos de chamadas associados às versões.

Podemos usar um átomo proposicional *remoteBackup* para indicar que agora o sistema possui um repositório remoto. Então, para indicar que o módulo do repositório remoto está sendo usado, podemos usar uma fórmula como:

$$\alpha_{out2} := @_5 AG(IN(\alpha_{in2} \wedge \alpha_{in3} \wedge \alpha_{in4}) \wedge remoteBackup)$$

Observe a estrutura da fórmula acima: temos uma parte que deve ser verificada nos grafos de chamadas, indicada pelo conectivo *IN*, e outra parte que deve ser analisada no vértice do grafo de versões.

Podemos assumir que parte dos dados armazenados no repositório consistem em planilhas que são armazenadas no formato CSV. Essas planilhas podem ser atualizadas, criadas ou deletadas por classes dos módulos X e Y, por exemplo. Suponha que o módulo Z é responsável por coletar essas planilhas em formato CSV e convertê-las para algum formato escolhido pelo usuário, como PDF. Neste cenário, uma propriedade como "Sempre que o usuário solicitar uma planilha em PDF, o método que converte a planilha para PDF é chamado" é obrigatória. A fórmula a seguir a formaliza:

$$\alpha_{in5} := AG((method :: selectFormat \wedge PDF) \rightarrow AF(method :: convertCsvToPdf))$$

Assumimos que existe um método, *selectFormat*, que permite que o usuário escolha o formato no qual deseja receber a planilha. O formato escolhido é PDF, indicado por um átomo proposicional. Podemos fazer isso para codificar informações extras. Por causa disso, usamos *method* para indicar os métodos, conforme explicamos anteriormente neste texto, para distinguir o que é método e o que é informação extra. Se isso acontece, então em todos os futuros possíveis, ou seja, em todas as possíveis execuções do programa, o método para converter de CSV para PDF é chamado.

Assumindo que inicialmente não existia opção para escolha de formato, podemos adicionar essa verificação a partir da versão na qual essa escolha foi incluída usando o conectivo @, conforme fizemos em α_{out2}. À medida que mais opções de formatos forem sendo incluídas, fórmulas com a mesma estrutura de α_{in5} podem ser utilizadas na verificação: é suficiente trocar PDF pelo formato de interesse.

O exemplo ilustrado aqui é apenas para dar uma ideia de como podemos aplicar o método que propomos em um caso real. Omitimos as classes que possuem os métodos em algumas fórmulas por simplicidade: elas não possuem relevância para o que estamos apontando. Acreditamos que esse exemplo mais destrinchado consiga passar ao leitor uma ideia melhor do uso do método de verificação que apresentamos neste trabalho.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, apresentamos uma metodologia de verificação de propriedades de sistemas por meio do método de Verificação de Modelos. A metodologia que propomos expressa as especificações de interesse utilizando lógicas que já demonstraram serem úteis na área e que são mais fáceis de compreender do que outras lógicas utilizadas nas demais abordagens, como lógica de primeira ordem. Até a presente análise, as fórmulas resultantes possuem tamanhos que as tornam fácil de serem lidas por seres humanos. Além disso, a metodologia adapta os algoritmos de Verificação de Modelos já existentes. Também definimos um novo operador para a lógica proposta.

Desenvolvemos uma metodologia com dois níveis de verificação. O nível 1 lida com uma versão específica do *software*. O nível 2 trata de propriedades globais relacionadas ao processo de desenvolvimento de *software* e engloba o nível 1. Para lidar com esses dois níveis de maneira homogênea, ligamos as estruturas usadas em cada um deles por meio de uma função e usamos o conectivo *IN*, proposto por este trabalho, para realizar a comunicação entre esses dois níveis nas fórmulas da lógica que desenvolvemos. A ideia dos dois níveis confere flexibilidade à metodologia de verificação. Podemos verificar uma série de especificações para as versões particulares no nível 1 e analisar como essas especificações se comportam ao longo do processo inteiro de desenvolvimento no nível 2.

Uma metodologia baseada em dois níveis de verificação, um local e um global, é a inovação deste trabalho. Ao aplicarmos essa metodologia no auxílio da Análise de Conformidade Arquitetural, procedimento essencial no desenvolvimento de sistemas de qualidade, demonstramos a sua utilidade e importância. Além disso, a metodologia que desenvolvemos funciona como um *framework* que pode ser ajustado para outros tipos de grafos que codifiquem aspectos diferentes do *software*, já que aqui lidamos com as relações de chamadas entre os métodos, e uma lógica apropriada, caso a lógica temporal que utilizamos no nível 1 não seja adequada. O algoritmo de Verificação de Modelos também necessitaria de mudanças, porém o mesmo esquema utilizado neste trabalho funcionaria: aplicamos calculamos a Verificação de Modelos no nível 1 e, com esses resultados, aplicamos a Verificação de Modelos no nível 2. Sendo assim, a metodologia que elaboramos neste texto funciona como um *framework*, o que aumenta a sua utilidade.

O algoritmo de Verificação de Modelos que sugerimos neste texto, adaptação dos algoritmos existentes na literatura para a Verificação de Modelos das lógicas de adotamos para

formar a nossa, é polinomial na entrada do problema, ou seja, na estrutura e na fórmula a ser avaliada. O polinômio que descreve a complexidade é de grau três no número de estados na estrutura de Kripke. Uma complexidade mais baixa é sempre desejável, porém a complexidade que obtemos aqui não é demasiadamente alta, sendo plausível na prática.

Quanto à implementação de um verificador de modelos, uma implementação do zero é mais simples do que modificar algum verificador já existente para a nossa lógica e nosso modelo. O grafo de versões e os grafos de chamadas não sofrem do problema da explosão de estados, comum para a Verificação de Modelos. Em geral, os modelos podem assumir tamanhos exponenciais na entrada, pois podemos ter um estado para cada conjunto no conjunto das partes dos átomos proposicionais (RYAN; HUTH, 2004). Entretanto, isso não acontece com a nossa metodologia proposta: o número de versões de um *software*, mesmo um que tenha estado na ativa por décadas, é muito menor do que os tamanhos de modelos exponenciais que acontecem no caso geral. Um fluxo de execução de um sistema também não tem um número demasiadamente grande de métodos a ponto de competir com modelos de tamanho exponencial. Logo, não precisamos lidar com a explosão de estados, o que facilita a implementação. Os verificadores de modelos também fazem otimizações nos algoritmos. Se deixarmos de lado tais otimizações, visto que já vimos que a complexidade do algoritmo de Verificação de Modelos é polinomial, a implementação fica ainda mais fácil.

Considerando os pontos que mencionamos acima, acreditamos que o método que desenvolvemos é uma ferramenta realmente útil para auxiliar a Análise de Conformidade Arquitetural, contribuindo para o desenvolvimento de sistemas confiáveis e de qualidade.

Analisamos cada versão específica de *software* de acordo com os grafos de chamadas, o que nos permite verificar propriedades relacionadas às classes e aos métodos, já que esses grafos capturam as relações de chamadas entre métodos. Para capturar outros aspectos de um *software* e verificar as especificações relacionadas a esses aspectos, precisamos de outras estruturas. Um exemplo são os grafos de fluxo de controle. Como trabalhos futuros, podemos analisar como utilizar esses grafos na Verificação de Modelos, representando-os como estruturas de Kripke, e elencar as propriedades que podemos analisar com base neles. É preciso averiguar se as lógicas temporais que usamos no atual trabalho são adequadas para as propriedades relacionadas aos grafos de fluxo de controle. Caso não sejam, precisamos encontrar uma lógica apropriada para ser usada no nível 1 de verificação.

Ainda no âmbito de trabalhos futuros, podemos investigar o uso de lógicas temporais de primeira ordem na especificação de propriedades. É possível que existam propriedades que necessitem dos quantificadores universal e existencial e da utilização de variáveis. Claro, teríamos que analisar o algoritmo de Verificação de Modelos para essas lógicas e calcular a sua complexidade.

Uma outra possibilidade de trabalho futuro é a melhoria do algoritmo de Verificação de Modelos para a lógica que propomos neste texto. Talvez seja possível usar alguma característica da lógica em questão ou das estruturas que utilizamos para desenvolver uma heurística que diminua a complexidade do algoritmo. Exige-se uma investigação mais profunda para termos certeza disso.

Um artigo explicando a metodologia que propomos neste trabalho foi aceito pelo Simpósio Brasileiro de Métodos Formais (SBMF) 2021 e publicado no Lecture Notes in Computer Science Volume 13130. O título do artigo é A Two-level Approach Based on Model Checking to Support Architecture Conformance Checking. Dados bibliográficos: (MENEZES *et al.*, 2021).

MENEZES, B.; MARTINS, A.T.; ROCHA, T. A. A two-level approach based on model checking to support architecture conformance checking. In: *Formal Methods: Foundations and Applications.* [S.l]: Stringer, 2021

REFERÊNCIAS

ALDRICH, J.; CHAMBERS, C.; NOTKIN, D. Archjava: Connecting software architecture to implementation. In: IEEE. **Proceedings of the 24th International Conference on Software Engineering. ICSE 2002**. [*S. l.*], 2002. p. 187–197.

BAIER, C.; KATOEN, J.-P. **Principles of Model Checking**. Estados Unidos da América: The MIT Press, 2008.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software architecture in practice**. [*S. l.*]: Addison-Wesley Professional, 2003.

BECKER-PECHAU, P. **Die stilbasierte Architekturprüfung: ein Ansatz zur Prüfung implementierter Softwarearchitekturen auf Architekturstil-Konformanz**. Tese (Doutorado) – Staats-und Universitätsbibliothek Hamburg Carl von Ossietzky, 2014.

Beyersdorff, O.; Meier, A.; Thomas, M.; Vollmer, H.; Mundhenk, M.; Schneider, T. Model checking CTL is almost always inherently sequential. In: **2009 16th International Symposium on Temporal Representation and Reasoning**. [*S. l.*: *s. n.*], 2009. p. 21–28.

BLACKBURN, P.; RIJKE, M. de; VENEMA, Y. **Modal Logic**. Cambridge University Press, 2002. (Cambridge Tracts in Theoretical Computer Science). ISBN 9780521527149. Disponível em: https://books.google.com.br/books?id=gFEidNVDWVoC.

BRUEGGE, B.; DUTOIT, A. H. **Object-Oriented Software Engineering Using UML, Patterns, and Java**. 3rd. ed. USA: Prentice Hall Press, 2009. ISBN 0136061257.

CARACCIOLO, A.; LUNGU, M. F.; NIERSTRASZ, O. A unified approach to architecture conformance checking. In: **2015 12th Working IEEE/IFIP Conference on Software Architecture**. [*S. l.*: *s. n.*], 2015. p. 41–50.

CHACON, S.; STRAUB, B. **Pro Git**. 2nd. ed. USA: Apress, 2014. ISBN 1484200772.

CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. **ACM Trans. Program. Lang. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 8, n. 2, p. 244–263, abr. 1986. ISSN 0164-0925. Disponível em: https://doi.org/10.1145/5397.5399.

CLARKE, E. M.; HENZINGER, T. A.; VEITH, H.; BLOEM, R. (Ed.). **Handbook of Model Checking**. Springer, 2018. ISBN 978-3-319-10574-1. Disponível em: https://doi.org/10.1007/978-3-319-10575-8.

CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. **Introduction to Algorithms**. MIT Press, 2009. (Computer science). ISBN 9780262533058. Disponível em: https://books.google.com.br/books?id=aefUBQAAQBAJ.

CQLINQ syntax. Disponível em: https://www.ndepend.com/docs/cqlinq-syntax/architect. Último acesso em: 23 de maio de 2021.

CZEPA, C.; TRAN, H.; ZDUN, U.; KIM, T. T. T.; WEISS, E.; RUHSAM, C. On the understandability of semantic constraints for behavioral software architecture compliance: A controlled experiment. In: **2017 IEEE International Conference on Software Architecture (ICSA)**. [*S. l.*: *s. n.*], 2017. p. 155–164.

DEISSENBOECK, F.; HEINEMANN, L.; HUMMEL, B.; JUERGENS, E. Flexible architecture conformance assessment with conqat. In: IEEE. **2010 ACM/IEEE 32nd International Conference on Software Engineering**. [*S. l.*], 2010. v. 2, p. 247–250.

DEITEL, P.; DEITEL, H. **Java How To Program - Early Objects**. 10th. ed. USA: Prentice Hall Press, 2014. ISBN 0133807800.

DUSZYNSKI, S.; KNODEL, J.; LINDVALL, M. Save: Software architecture visualization and evaluation. In: IEEE. **2009 13th European conference on software maintenance and reengineering**. [*S. l.*], 2009. p. 323–324.

EBBINGHAUS, H.-D.; FLUM, J.; THOMAS, W.; FEREBEE, A. S. **Mathematical logic**. [*S. l.*]: Springer, 1994.

FILHO, J. de L. M. **Uma Solução para Verificação Estática de Conformidade Arquitetural do Tratamento de Exceção**. Dissertação (Mestrado) – Universidade Federal do Ceará, 2016.

FISHER, M. **An Introduction to Practical Formal Methods Using Temporal Logic**. 1st. ed. [*S. l.*]: Wiley Publishing, 2011. ISBN 0470027886.

GROVE, D.; CHAMBERS, C. A framework for call graph construction algorithms. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM New York, NY, USA, v. 23, n. 6, p. 685–746, 2001.

HEROLD, S. **Architectural compliance in component-based systems**. Tese (Doutorado) – Clausthal University of Technology, 2011.

HOFFMANN, A. B. G. *et al.* Um modelo para o controle de versões de sistemas para apoio ao desenvolvimento colaborativo através da web. Florianópolis, SC, 2002.

HOU, D.; HOOVER, H. J. Using SCL to specify and check design intent in source code. **IEEE Transactions on Software Engineering**, IEEE, v. 32, n. 6, p. 404–423, 2006.

IADAROLA, G.; MARTINELLI, F.; MERCALDO, F.; SANTONE, A. Call graph and model checking for fine-grained android malicious behaviour detection. **Applied Sciences**, Multidisciplinary Digital Publishing Institute, v. 10, n. 22, p. 7975, 2020.

IEEE, I. . Iso/iec standard for systems and software engineering - recommended practice for architectural description of software-intensive systems. **ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15**, p. 1–24, 2007.

JQASSISTANT. Disponível em: https://jqassistant.org/architect. Último acesso em: 23 de maio de 2021.

LATTIX architect. Disponível em: https://www.lattix.com/products-architecture-issues/architect. Último acesso em: 23 de maio de 2021.

MACKER. Disponível em: https://innig.net/macker/. Último acesso em: 23 de maio de 2021.

MENEZES, B.; MARTINS, A. T.; ROCHA, T. A. A two-level approach based on model checking to support architecture conformance checking. In: **Formal Methods: Foundations and Applications**. [*S. l.*]: Springer, 2021.

MENS, K.; MENS, T. Codifying high-level software abstractions as virtual classifications. **Position paper, Programming Technology Lab, Vrije Universiteit Brussel**, 2000.

MITSCHKE, R.; EICHBERG, M.; MEZINI, M.; GARCIA, A.; MACIA, I. Modular specification and checking of structural dependencies. In: **Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development**. New York, NY, USA: Association for Computing Machinery, 2013. (AOSD '13), p. 85–96. ISBN 9781450317665. Disponível em: https://doi.org/10.1145/2451436.2451448.

MOOR, O. D.; SERENI, D.; VERBAERE, M.; HAJIYEV, E.; AVGUSTINOV, P.; EKMAN, T.; ONGKINGCO, N.; TIBBLE, J. .QL: Object-oriented queries made easy. In: SPRINGER. **International Summer School on Generative and Transformational Techniques in Software Engineering**. [*S. l.*], 2007. p. 78–133.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. Software reflexion models: Bridging the gap between source and high-level models. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, New York, NY, USA, v. 20, n. 4, p. 18–28, out. 1995. ISSN 0163-5948. Disponível em: https://doi.org/10.1145/222132.222136.

PASSOS, L.; TERRA, R.; VALENTE, M. T.; DINIZ, R.; MENDONçA, N. Static architecture-conformance checking: An illustrative overview. **IEEE Software**, v. 27, n. 5, p. 82–89, 2010.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 4, p. 40–52, out. 1992. ISSN 0163-5948. Disponível em: https://doi.org/10.1145/141874.141884.

PRESTON-WERNER, T. **Semantic Versioning 2.0.0**. Disponível em: https://semver.org/spec/v2.0.0.html. Acesso em: 13 de maio de 2021.

PRUIJT, L. J.; KöPPE, C.; WERF, J. M. van der; BRINKKEMPER, S. Husacct: Architecture compliance checking with rich sets of module and rule types. In: **Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (ASE '14), p. 851–854. ISBN 9781450330138. Disponível em: https://doi.org/10.1145/2642937.2648624.

QUEIROZ, I.; BUSTAMANTE, L.; MARTINS, A.; ALCANTARA, J. Reasoning on ontology version space with temporal logics. In: . [*S. l.*: *s. n.*], 2014.

RYAN, M.; HUTH, M. **Logic in Computer Science: Modelling and Reasoning about Systems**. Estados Unidos da América: Cambridge University Press, 2004.

SCHRÖDER, S. **Ontology-Based Architecture Enforcement: Defining and Enforcing Software Architecture as a Concept Language using Ontologies and a Controlled Natural Language**. Tese (Doutorado) – Staats-und Universitätsbibliothek Hamburg Carl von Ossietzky, 2020.

SILVA, L. de; BALASUBRAMANIAM, D. Controlling software architecture erosion: A survey. **J. Syst. Softw.**, Elsevier Science Inc., USA, v. 85, n. 1, p. 132–151, jan. 2012. ISSN 0164-1212. Disponível em: https://doi.org/10.1016/j.jss.2011.07.036.

SOMMERVILLE, I. **Engenharia de software**. PEARSON BRASIL, 2011. ISBN 9788579361081. Disponível em: https://books.google.com.br/books?id=H4u5ygAACAAJ.

SONARGRAPH-ARCHITECT. 2017.

STRUCTURE 101. Disponível em: https://structure101.com/. Último acesso em: 23 de maio de 2021.

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

UBAYASHI, N.; NOMURA, J.; TAMAI, T. Archface: a contract place where architectural design and code meet together. In: **2010 ACM/IEEE 32nd International Conference on Software Engineering**. [*S. l.*: *s. n.*], 2010. v. 1, p. 75–84.

VARDI, M.; WILKE, T. Automata: from logics to algorithms. In: . [*S. l.*: *s. n.*], 2008. p. 629–736.

WEST, D. B. **Introduction to Graph Theory**. 2. ed. [*S. l.*]: Prentice Hall, 2000. ISBN 0130144002.

APÊNDICE A – CORRETUDE DE FÓRMULAS

Apresentamos a seguir as provas da corretude das fórmulas apresentadas na Seção 4.4, que formalizam algumas especificações relacionadas ao padrão de arquitetural conhecido como arquitetura de camadas (Exemplo 2.2). Seguimos a mesma ordem utilizada na apresentação das especificações na Seção 4.4. Não repetimos o significado das fórmulas novamente aqui. Em caso de dúvidas, sugerimos que o leitor retorne ao Exemplo 4.4.

1. *EF(A ∧ EX(B))*: a classe *A* chama diretamente a classe *B*.

Apresentamos a prova de que a fórmula proposta realmente captura o que procuramos, i.e, a estrutura de Kripke satisfaz à fórmula proposta se e somente se a situação exibida na Figura 23 acontece na estrutura.

Prova: sejam M uma estrutura de Kripke e s um estado inicial qualquer. Vamos mostrar que M,s satisfaz $EF(A \wedge EX(B))$ se e somente se existe um estado com o átomo proposicional A no rótulo e que tem como sucessor um estado com o átomo proposicional B no rótulo, o que corresponde à situação que buscamos. Aplicamos a semântica da CTL para analisar quando a fórmula proposta é satisfeita:

$M, s \models EF(A \wedge EX(B))$ se e somente se

Existe um caminho $s \rightarrow s_2 \rightarrow s_3 \rightarrow ...$ tal que M, $s_i \models$ A $\wedge$ EX(B) para algum $s_i \in \{s, s_2, s_3, ...\}$ se e somente se

$M, s_i \models A$ e $M, s_i \models EX(B)$ se e somente se

$A \in L(s_i)$ e existe s_j tal que $s_i \rightarrow s_j$, ou seja, s_j é um sucessor de s_i, e $B \in L(s_j)$.

Essa é exatamente a situação que queremos encontrar na estrutura de Kripke, conforme a explicação dada anteriormente. Logo, a fórmula proposta captura de fato a consulta em questão.

C.Q.D

2. $EF(A \wedge EX(\neg B \wedge EF(B)))$: a classe A chama indiretamente a classe B.

Seguindo o que fizemos no caso anterior, apresentamos a prova da corretude da fórmula proposta.

Prova: sejam M uma estrutura de Kripke e s um estado inicial qualquer. Vamos mostrar que M, s satisfaz $EF(A \wedge EX(\neg B \wedge EF(B)))$ se e somente se existe um estado v com A no rótulo e que tem pelo menos um sucessor u que não tem B na sua rotulação, porém, a partir de u, podemos alcançar outro estado w que possui B no seu rótulo. É a situação apresentada na Figura 24. Aplicando a semântica da CTL, segue que:

$M, s \models EF(A \wedge EX(\neg B \wedge EF(B)))$ se e somente se

Existe um caminho $s \rightarrow s_2 \rightarrow s_3 \rightarrow ...$ tal que $M, s_i \models A \wedge EX(\neg B \wedge EF(B))$ para algum $s_i \in \{s, s_2, s_3, ...\}$ se e somente se

$M, s_i \models A$ e $M, s_i \models EX(\neg B \wedge EF(B))$ se e somente se

$A \in L(s_i)$ e existe um s_j tal que $s_i \rightarrow s_j$, ou seja, s_j é um sucessor de s_i, e $M, s_j \models \neg B \wedge EF(B)$ se e somente se

$A \in L(s_i)$, $M, s_j \models \neg B$ e $M, s_j \models EF(B)$ para um tal M, s_j se e somente se

$A \in L(s_i)$, $B \notin L(s_j)$ e existe um caminho $s_j \rightarrow s_{j2} \rightarrow s_{j3} \rightarrow ...$ tal que $M, s_{jk} \models B$ para algum $s_{jk} \in \{s_{j2}, s_{j3}, ...\}$ se e somente se

$A \in L(s_i)$, $B \notin L(s_j)$ e $B \in L(s_{jk})$, onde s_j é um sucessor de s_i e s_{jk} é um descendente de s_j. Note que s_j e s_{jk} são necessariamente distintos, pois se fossem iguais, teríamos um estado que satisfaz $B \wedge \neg B$, o que é uma contradição.

Essa é a situação que esperamos encontrar na estrutura de Kripke para responder positivamente à consulta em questão. Logo, a fórmula proposta está correta.

C.Q.D

3. $EF(m \wedge EXEF(n))$: existe uma sequência de chamadas partindo do método m que alcança o método n.

Apresentamos a corretude da fórmula proposta a seguir.

Prova: sejam M uma estrutura de Kripke e s um estado inicial qualquer. Vamos mostrar que M, s satisfaz $EF(m \wedge EXEF(n))$ se e somente existe um estado u com m no seu rótulo e que, a partir de um sucessor de u chegamos em um estado com n no seu rótulo. Não tem problema se esse estado já for o sucessor de u. Pela maneira como a estrutura de Kripke é construída, essa situação corresponde exatamente a uma sequência de chamadas partindo de m que alcança n.

$M, s \models EF(m \wedge EXEF(n))$ se e somente se

Existe um caminho $s \rightarrow s_2 \rightarrow s_3 \rightarrow ...$ tal que $M, s_i \models m \wedge EXEF(n)$ se e somente se

$M, s_i \models m$ e $M, s_i \models EXEF(n)$ se e somente se

$m \in \mathrm{L}(s_i)$ e existe s_j tal que $s_i \rightarrow s_j$, ou seja, s_j é um sucessor de s_i, tal que $M, s_j \models EF(n)$ se e somente se

$m \in \mathrm{L}(s_i)$ e existe um caminho $s_j \rightarrow s_{j2} \rightarrow s_{j3}...$ tal que $s_k \models n$ para algum $s_k \in \{s_j, s_{j2}, s_{j3}...\}$ se e somente se

$m \in \mathrm{L}(s_i)$ e $n \in L(s_k)$ para um tal s_k.

Logo, a fórmula sugerida é satisfeita se e somente se temos um estado s_i com m no seu rótulo e um estado s_k com n no seu rótulo aparece em um caminho que começa em s_i. Essa é exatamente a situação que esperamos encontrar na estrutura de Kripke para responder positivamente à consulta em questão. Logo, a fórmula está correta.

C.Q.D

4. *EF(m ∧ EX EX EF(m))*: existe uma sequência de chamadas partindo do método *m* que alcança o método *m*.

Prova: a prova da corretude da fórmula sugerida é exatamente como a prova do caso anterior, porém trocamos o átomo proposicional *n* por *m*.

C.Q.D

5. $EF(A \wedge EX(EGEF(A)))$: a classe *A* está em um laço de chamadas.

Um método de *A* é chamado. Após essa chamada, algum método de *A* é sempre chamado novamente enquanto o programa continuar a ser executado. Procuramos no grafo de chamadas pela situação apresentada na Figura 25. A prova da corretude da fórmula sugerida é dada a seguir.

Prova: sejam *M* um estrutura de Kripke e *s* um estado inicial qualquer. Vamos mostrar que M, s satisfaz $EF(A \wedge EX(EGEF(A)))$ se e somente se existe um estado com *A* no seu rótulo e a partir desse estado chegamos indefinidamente a estados com *A* nos seus rótulos.

$M, s \models EF(A \wedge EX(EGEF(A)))$ se e somente se

Existe um caminho $s \rightarrow s_2 \rightarrow s_3 \rightarrow ...$ tal que $M, s_i \models A \wedge$ EX(EGEF(A)) para algum $s_i \in \{s, s_2, s_3, ...\}$ se e somente se

$M, s_i \models A$ e $M, s_i \models EX(EGEF(A))$ se e somente se

$A \in$ L(s_i) e existe um s_j tal que $s_i \rightarrow s_j$, ou seja, s_j é um sucessor de s_i, e $M, s_j \models EGEF(A)$ se e somente se

$A \in L(s_i)$ e existe um caminho $s_j \rightarrow s_{j2} \rightarrow s_{j3} \rightarrow ...$ temos que $M, s_{jk} \models EF(A)$ para todo $s_{jk} \in \{s_j, s_{j2}, s_{j3}, ...\}$ se e somente se

$A \in$ L(s_i) e existe um caminho $s_{jk} \rightarrow s_{jk2} \rightarrow s_{jk3} \rightarrow ...$ a partir de todo $s_{jk} \in \{s_j, s_{j2}, s_{j3}, ...\}$ tal que $M, s_{jkk} \models A$ para algum $s_{jkk} \in \{s_{jk}, s_{jk2}, s_{jk3}, ...\}$ se e somente se

$A \in$ L(s_i) e $A \in L(s_{jkk})$ para tais s_{jkk}.

Isso mostra que a fórmula proposta é satisfeita se e somente se encontramos um estado s_i com *A* no seu rótulo e, a partir de s_i, encontramos um caminho tal que a partir de todos

os estados neste caminho (por causa do operador EG) é possível alcançar um vértice com A no rótulo. Com isso, concluímos que temos um estado com A no seu rótulo e, começando neste estado, sempre chegamos novamente a um estado (que pode ser o próprio s_i, basta que esse tenha uma transição para si mesmo) com A no rótulo. Somente isso não prova que a classe A está em um *loop* de chamadas. Porém, lembre-se que a estrutura de Kripke que estamos lidando é finita e estamos trabalhando com caminhos infinitos. Logo, um caminho infinito irá eventualmente entrar em um ciclo. Portanto, o caminho $s_j \rightarrow s_{j2} \rightarrow s_{j3} \rightarrow ...$ no qual todos os estados satisfazem $EF(A)$ da demonstração acima certamente contém um ciclo, no qual repete infinitamente um mesmo conjunto de estados. A partir destes estados, temos outros caminhos com pelo menos um estado que satisfaz A. Esses outros caminhos também possuem ciclos, pelo menos motivo que acabamos de explicar. Portanto, caso a fórmula que propomos seja satisfeita, existe um ciclo de chamadas no qual um método da classe A sempre volta a aparecer. Com isso, podemos concluir que a classe A está em um *loop* de chamadas.

C.Q.D

APÊNDICE B – ESTADOS QUE ESTÃO EM $F^{N+1}(\emptyset)$

Na Seção 4.5, ao provar a corretude do algoritmo de Verificação de Modelos para os conectivos do passado, precisamos demonstrar quais são os estados que estão na $(n+1)$-ésima aplicação da função F a partir do conjunto vazio. Apresentamos aqui uma prova por indução de quais são os estados presentes em $F^{n+1}(\emptyset)$, para $F(X) = [\phi] \cup next_{\exists}(X)$. As definições dos elementos dessa função já foram apresentados na Seção 4.5.

Prova: $F^{n+1}(\emptyset)$ possui todos os estados que satisfazem ϕ e todos os estados que estão em caminhos de tamanho n que partem dos estados que satisfazem ϕ.

Caso base: n = 0.

Temos:

$F^{0+1}(\emptyset) = F(\emptyset) = [\phi] \cup next_{\exists}(\emptyset) = [\phi]$, pois o conjunto vazio não tem sucessores.

Um caminho de tamanho 0 possui um único estado. Logo, caso esse estado satisfaça ϕ, ou seja, caso ele esteja em $[\phi]$, temos que $F^{0+1}(\emptyset)$ possui todos os estados que satisfazem ϕ e todos os estados que estão em caminhos de tamanho 0 que partem dos estados que satisfazem ϕ. Note que essa propriedade continua válida mesmo se o estado não satisfizer ϕ.

Hipótese de indução: temos que para $k \geq 1$ vale que $F^{k+1}(\emptyset)$ possui todos os estados que satisfazem ϕ e todos os estados que estão em caminhos de tamanho k que partem dos estados que satisfazem ϕ.

Passo de indução: mostramos que a propriedade vale para $k+1$.

$F^{(k+1)+1}(\emptyset) = F(F^{k+1}(\emptyset)) = [\phi] \cup next_{\exists}(F^{k+1}(\emptyset))$

Pela hipótese de indução, temos que $F^{k+1}(\emptyset)$ possui todos os estados que satisfazem ϕ e todos os estados que estão em caminhos de tamanho k que partem dos estados que satisfazem ϕ. Temos que $next_{\exists}(F^{k+1}(\emptyset))$ adiciona os estados que são sucessores dos estados em $F^{k+1}(\emptyset)$.

Logo, o tamanho dos caminhos é aumentado em 1. Consequentemente, $F^{(k+1)+1}(\emptyset)$ possui todos os estados que satisfazem ϕ e todos os estados que estão em caminhos de tamanho $k+1$ que partem dos estados que satisfazem ϕ.

Concluímos que $F^{n+1}(\emptyset)$ possui todos os estados que satisfazem ϕ e todos os estados que estão em caminhos de tamanho n que partem dos estados que satisfazem ϕ, como queríamos demonstrar.

C.Q.D