UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

IURI FERNANDES QUEIROZ

VERSIONAMENTO DE ONTOLOGIAS BASEADO EM LÓGICAS TEMPORAIS

FORTALEZA, CEARÁ

2012

IURI FERNANDES QUEIROZ

VERSIONAMENTO DE ONTOLOGIAS BASEADO EM LÓGICAS TEMPORAIS

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Área de concentração: Lógica e Inteligência Artificial

Orientador: Profa. Dra. Ana Teresa de Castro Martins

Co-Orientador: Prof. Dr. João Fernando de Lima Alcântara

FORTALEZA, CEARÁ

2012

A000z QUEIROZ, I. F..

Versionamento de Ontologias Baseado em Lógicas Temporais / Iuri Fernandes Queiroz. 2012.

53p.;il. color. enc.

Orientador: Profa. Dra. Ana Teresa de Castro Martins

Coorientador: Prof. Dr. João Fernando de Lima Alcântara

Dissertação(Ciência da Computação) - Universidade Federal do Ceará, Departamento de Computação, Fortaleza, 2012.

1. Versionamento de Ontologias 2. *Model Checking* 3. Complexidade Computacional I. Profa. Dra. Ana Teresa de Castro Martins(Orient.) II. Universidade Federal do Ceará– Ciência da Computação(Mestrado) III. Mestre

CDD:000.0

IURI FERNANDES QUEIROZ

VERSIONAMENTO DE ONTOLOGIAS BASEADO EM LÓGICAS TEMPORAIS

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação, da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação. Área de concentração: Lógica e Inteligência Artificial

Aprovada em: __/__/____

BANCA EXAMINADORA

Profa. Dra. Ana Teresa de Castro Martins
Universidade Federal do Ceará - UFC
Orientador

Prof. Dr. João Fernando de Lima Alcântara
Universidade Federal do Ceará - UFC
Coorientador

Prof. Dr. Carlos Eduardo Fisch de Brito
Universidade Federal do Ceará - UFC

Prof. Dr. Mario Roberto Folhadela Benevides
Universidade Federal do Rio de Janeiro - UFRJ

Aos meus Pais.

AGRADECIMENTOS

"*Fear can't hurt you any more than a dream.*"

(William Golding)

RESUMO

As Ontologias são uma forma de representar conhecimento muito usadas na WEB Semântica. Com o passar do tempo, elas são modificadas e para manter compatibilidade com aplicações que as usam, são criadas versões dessa ontologia. Dado um conjunto de versões de uma ontologia, encontrar qual a melhor versão para uma aplicação não é uma tarefa trivial. Esse problema torna-se mais complexo quando temos uma forma de desenvolvimento sem uma entidade reguladora, que tende a ser comum com a popularização do uso de Ontologias.

Nós propomos uma representação para o histórico do desenvolvimento de uma ontologia sobre um ponto de vista da análise de suas versões. Nossa representação abrange tanto a forma de desenvolvimento mais usual quanto a sem uma entidade reguladora, estendendo a abordagem anterior obtida por Huang e Stuckenschmidt. Para analisar o histórico das versões descrito na nossa representação, construímos uma família de lógicas que associam uma Linguagem Modal Temporal a uma Linguagem de Consulta em Ontologias. Estudamos as expressividades de algumas dessas lógicas e mostramos ainda uma forma geral para obter a Complexidade Computacional do problema de *Model Checking* dessas lógicas.

Palavras-chave: Versionamento de Ontologias. *Model Checking*. Complexidade Computacional.

ABSTRACT

Ontologies are a way to represent knowledge which is very popular in the Semantic WEB initiative. As the time passes ontologies may be modified. To mantain the compatibility with applications that use a ontology, versions of it are created. Given an ontology version set, to find out which version is the best for an application is not a trivial task. This problem becomes more complex when we have a development withtout a regulatory entity which tends to be more common with Ontology increasing popularity.

We propose a representation for the history of a ontology development on a point of view of its versions analisys. Our representation covers the usual form of development as well as the form without a regulatory entity, extending previous approachs by Huang and Stuckenschmidt. To analyse the versions history described in our representation, we build a family of logics which associate a Temporal Modal Language and a Query Language. We also show a general form to obtain Computational complexity results for the Model Checking problem of one such logic.

Keywords: Ontology Versionning. Model Checking. Computational Complexity.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização do Trabalho

Uma parte fundamental da realização da Web Semântica [Berners-Lee, Hendler e Lassila 2001] é a representação das ontologias. Nesse contexto, ontologias são representações formais de um domínio de conhecimento de interesse. Elas tem um papel essencial na Web Semântica por ser a parte responsável por representar os conceitos do mundo real em uma linguagem formal. A criação de linguagens com esse propósito é uma importante área de estudo, pois elas devem prover facilidade de representação e meios de raciocinar sobre uma ontologia.

Uma vez representadas, muitos fatores podem requerer que ontologias sejam modificadas. Um desses é a mudança de visão do que importa em um domínio. Isso pode acontecer quando se percebe que o que foi representado não é bem o que se deseja. Outro fator é o acesso a informações antes desconhecidas. A obtenção de novas informações, pode acarretar em inconsistências com o que tinha sido representado na ontologia. Como o domínio que as ontologias representam pode ser mutável, elas devem ser capazes de representar as mudanças ocorridas. O campo responsável por estudar como modificá-las é o de Mudança de Ontologias [Flouris et al. 2008].

Após feita uma modificação, para manter compatibilidade entre aplicações que usavam a ontologia antes da modificação, frequentemente, ela continua sendo disponibilizada sem alterações. Para disponibilizar a ontologia modificada, é criada uma versão. Assim, diversas versões de uma mesma ontologia são mantidas. Como lidar com essas versões é estudado no campo de Mudança de Ontologias pela subárea de Versionamento de Ontologias [Klein e Fensel 2001].

No processo de versionamento, uma vez efetuada uma modificação, não há garantias de que a nova versão criada seja útil às aplicações e ontologias que usavam a versão anterior. Essa situação torna-se mais extrema quando lidamos com ontologias desenvolvidas sem entidades que garantam que haja só uma versão mais recente de uma ontologia. Por exemplo, suponha que dois desenvolvedores elaborem, separadamente, modificações de uma mesma ontologia. Quando lançarem suas versões, haverá a possibilidade de escolha entre essas duas por parte do usuário dessa ontologia. Esse exemplo ilustra uma situação que tende a ser mais frequente com a implementação da Web Semântica.

1.2 Objetivos e Relevância

Nesse trabalho, temos por objetivo disponibilizar um formalismo para lidar com múltiplas modificações de uma ontologia, disponibilizando meios de comparação de versões de ontologias desenvolvidas com a presença ou ausência de entidades reguladores. Assim, primeiramente analisaremos a área de Mudança de Ontologias para definir melhor o escopo do trabalho. Isso será realizado no Capítulo 3, onde serão examinados os tipos de mudança de uma ontologia, em especial o Versionamento de Ontologias.

Estudaremos o Versionamento de Ontologias sob um aspecto temporal, vendo as versões como representações do domínio de interesse de uma ontologia válidas em um certo instante. Para isso, nossa metodologia será o uso de Lógicas Temporais, que são lógicas com operadores modais que fornecem meios para analisar a evolução de sistemas. No Capítulo 4 estudaremos Lógicas Modais. Veremos, primeiramente, os conceitos básicos de Lógicas modais. Ainda nesse capítulo, estudaremos Lógicas Temporais que são lógicas modais mais adequadas para formalizar noções de tempo. Por último, mostraremos extensões de Lógicas Temporais com operadores de Lógica Híbrida, que, por sua vez, é uma extensão da Lógica Modal adicionando operadores especiais.

Por último, no Capítulo 5 construiremos o nosso formalismo, mostrando exemplos da sua utilidade. Para isso, usaremos as Lógicas estudadas no capítulo 4 como base para a formalização. Estudaremos, ainda, a complexidade da nossa construção para as diversas lógicas mostradas.

Nosso trabalho apresenta um formalismo original com complexidade computacional bem definida para comparação entre versões de uma ontologia, mesmo que não haja uma entidade reguladora do desenvolvimento dessa ontologia. Alguns esforços para resolver um problema parecido foi feito em [Huang e Stuckenschmidt 2005], mas ontologias que não tem entidades reguladoras não são abordadas.

2 NOÇÕES DE COMPLEXIDADE COMPUTACIONAL

Nesse trabalho, estaremos muito interessados na complexidade computacional dos problemas que lidamos, com o objetivo de avaliar a aplicabilidade das soluções propostas. Para isso, faremos um breve estudo da área. Referências básicas que usaremos para explorar esse campo de estudo são [Papadimitriou 1994, Arora e Barak 2009]. Na seções 2.1 e 2.2 definiremos problemas de decisão e o modelo de computação utilizado. Finalmente, na seção 2.3 definiremos Classes de Complexidade e mostraremos algumas classes importantes.

2.1 Problemas de Decisão

Um problema computacional pode ser visto como uma pergunta a respeito de um conjunto de dados de entrada. Deve-se processar uma entrada, obtendo uma saída, que é a resposta do problema para essa entrada. Mais precisamente, chamamos uma determinada entrada de instância de um problema. Um tipo importante de problema é o de decisão. Esses problemas tem as saídas possíveis restritas aos valores **SIM** e **NÃO** para cada instância. Uma definição mais formal é apresentada em [Arora e Barak 2009] e mostrada a seguir.

Definição 2.1.1. *Seja uma função* $f : \{0,1\}^* \mapsto \{0,1\}$*, dizemos que* f *é uma função booleana e que* $L_f = \{x : f(x) = 1\}$ *é uma* linguagem *ou um* problema de decisão*. Computar a função* $f(x)$ *é o mesmo de decidir se* $x \in L_f$ *ou resolver o problema de decisão em questão para a entrada* x*.*

Diz-se que um problema de decisão é decidível ou que tem a propriedade de decidibilidade se, para qualquer entrada, é possível computar $f(x)$ para qualquer $x \in \{0,1\}^*$. Ou seja, se existe um procedimento bem definido que sempre chega à resposta correta, qualquer que seja a entrada.

Os problemas de decisão também podem ser vistos como partições do conjunto de instâncias, onde cada partição contém as instâncias cuja mesma resposta é obtida para o problema em questão. Em particular, os problemas de decisão definem duas partições. Nos referimos às possíveis respostas de um problema de decisão com **SIM** para o valor 1 e **NÃO** para o valor 0. Frequentemente, chamamos a partição que contém as instâncias cuja resposta é **SIM** de conjunto das instâncias **SIM**. O conjunto das instâncias **NÃO** é o complemento do conjunto das instâncias **NÃO**.

2.2 Modelo de Computação

Para resolver um problema, é necessário um modelo formal da atividade de computar. As alternativas mais famosas são as funções recursivas e as máquinas de Turing(MT), porém existem muitos outros como o cálculo *lambda*. A tese de Church-Turing mostra a equivalência desses modelos. Sabemos, também, que é possível simular qualquer modelo de computação

conhecido usando uma máquina de Turing, mas com alguma perda de eficiência. A única exceção é o modelo quântico, o qual não se sabe se apresenta maior poder de computação que a máquina de Turing.

A máquina de Turing é um formalismo matemático que apresenta regras simples de manipulação de cadeias de um dado alfabeto. A seguir, mostraremos a definição formal dessas máquinas como apresentado em [Papadimitriou 1994].

Definição 2.2.1 (Máquina de Turing)**.** *Uma máquina de Turing é uma quadrupla* $M = (K, \Sigma, \delta, s)$. *Aqui* K *é um conjunto finito de estados;* $s \in K$ *é o estado inicial.* Σ *é o conjunto de símbolos (dizemos que* Σ *é o alfabeto de* M*). Assumimos que* K *e* Σ *são conjuntos disjuntos.* Σ *sempre apresenta os símbolos* $\sqcup$ *e* $\triangleright$*: o símbolo de branco e o primeiro símbolo respectivamente. Finalmente,* δ *é uma função de transição, a qual mapeia* $K \times \Sigma$ *em* $(K \cup \{h, \text{“sim”}, \text{“não”}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. *Assumimos que* h *(estado de parada), “sim” (estado de aceitação), “não” (estado de rejeição) e os símbolos de direção* $\leftarrow$ *para “esquerda”,* $\rightarrow$ *para “direita”,* $-$ *para “fique” não estão em* $K \cup \Sigma$.

A parte principal da definição anterior é δ, pois conterá o comportamento da máquina. Podemos pensar uma MT como um dispositivo que lê um símbolo de cada vez de uma fita infinita para dois lados. Dependendo do símbolo lido e do estado em que a máquina se encontra, δ determina para qual estado a MT deve ir, o símbolo que deve ser escrito e o movimento a ser executado. Para a execução de uma máquina M com uma determinada entrada x, M começa lendo da fita o símbolo $\triangleright$ e à direita desse símbolo encontram-se a entrada x descrita símbolo por símbolo e ao final um número infinito de espaços em branco. Para cada transição de δ executada, dizemos que a M executou um passo. Por último a máquina pode terminar sua execução, alcançando algum dos estados h, “sim” e “não”. Dizemos que M aceita x se M com entrada x termina sua execução no estado “sim” ($M(x)$ = “sim”) e que rejeita se M com entrada x termina no estado “não” ($M(x)$ = “não”). Dizemos que M computa uma linguagem L se para $x \in L$, $M(x)$ =“sim” e se $x \notin L$, $M(x)$ =“não”.

Ao definirmos que δ é uma função, restringimos a nossa MT para o que chamamos de Máquina de Turing *determinística*. Pode-se relaxar essa restrição e permitir que δ seja uma relação. Uma máquina com uma relação de transição é chamada de máquina de Turing *não-determinística*. Com esse tipo de máquina, em um dado momento da computação, podemos seguir mais de um caminho, dependendo da tupla da relação de transição utilizada. Passamos a ter o que chamamos de árvore de computação, onde uma cadeia é aceita se existe pelo menos um caminho que chega ao estado “sim”.

2.3 Classes de Complexidade

Ao definirmos problemas computacionais, uma das primeiras ideias que podem passar à mente do leitor é como comparar suas dificuldades. Isso é feito através da quantificação do uso de recursos computacionais necessários para resolvê-los em uma determinada máquina. Os recursos mais utilizados para isso são tempo e espaço ou memória. Através dessas medidas,

os problemas são agrupados em classes de complexidade. Uma classe de complexidade é definida por Arora e Barack em [Arora e Barak 2009] como "um conjunto de funções que podem ser computadas em uma determinada máquina satisfazendo uma dada limitação de recurso", ou seja, cada classe agrupará diversos problemas que podem ser resolvidos com a mesma restrição de recursos.

Algumas classes de complexidade podem ser usadas para definir precisamente muitas outras classes. Definiremos a seguir algumas delas: **DTIME**, **NTIME** e **SPACE**.

Definição 2.3.1. *Seja uma função $T : \mathbb{N} \mapsto \mathbb{N}$. Uma linguagem L está em **DTIME**$(T(n))$ se e somente se existe uma máquina de Turing determinística que roda em tempo $c \cdot T(n)$ para alguma constante $c > 0$ e decide L.*

Definição 2.3.2. *Seja uma função $T : \mathbb{N} \mapsto \mathbb{N}$. Uma linguagem L está em **NTIME**$(T(n))$ se e somente se existe uma máquina de Turing não-determinística que roda em tempo $c \cdot T(n)$ para alguma constante $c > 0$ e decide L.*

Definição 2.3.3. *Seja uma função $S : \mathbb{N} \mapsto \mathbb{N}$ e $L \subseteq \{0,1\}^*$. Dizemos que $L \in$ **SPACE**$(s(n))$ se existe uma constantes c e uma máquina de Turing determinística M que decide L tal que no máximo $c \cdot s(n)$ posições da fita de M, excluindo as posições da entrada, são visitados pela cabeça de M para qualquer entrada de tamanho n.*

A seguir, mostraremos classes importantes as quais aparecerão no decorrer do trabalho e que podem ser definidas com base nas classes anteriores.

- **PTIME** $= \bigcup_{c \geq 1}$ **DTIME**(n^c)
- **NPTIME** $= \bigcup_{c \geq 1}$ **NTIME**(n^c)
- **EXPTIME** $= \bigcup_{c \geq 1}$ **DTIME**(2^{n^c})
- **NEXPTIME** $= \bigcup_{c \geq 1}$ **NTIME**(2^{n^c})
- **PSPACE** $= \bigcup_{c \geq 0}$ **SPACE**(n^c)

É usual na literatura acharmos as denominações **P**, **NP**, **EXP** e **NEXP** para indicar **PTIME**, **NPTIME**, **EXPTIME**, **NEXPTIME** respectivamente.

Como vimos acima, classes de complexidade diferentes podem ser obtidas quando variamos a restrição de recursos. Intuitivamente, classes com uma maior restrição de recursos, ou seja, em que tempo ou espaço são menores, contém problemas que podem ser resolvidos mais eficientemente. Usualmente, consideramos que a classe **PTIME** contém os problemas que podem ser resolvidos eficientemente. Apesar de bastante tradicional, essa noção de eficiência deve ser usada com cautela, pois quando a constante c é grande, n^c pode crescer muito rapidamente, apresentando um comportamento semelhante a uma função exponencial para muitas entradas.

Uma importante noção é a de um problema ser completo para uma classe de complexidade.

Definição 2.3.4. *Seja A um problema de decisão, C uma classe de complexidade e $\leq$ uma relação de redução. Dizemos que A é C-difícil via a redução $\leq$ se, para todo $B \in C$, $B \leq A$. Se for A for C-difícil e $A \in C$ então dizemos que A é C-completo.*

2.3.1 Classes contidas em PTIME

Apesar da classe **PTIME** ser a classe dos problemas que podemos chamar de fáceis, há graus de dificuldade desses problemas. Logo, existem várias classes contidas em **PTIME**. Definiremos a seguir algumas classes importantes.

Definição 2.3.5. *Para todo d, uma linguagem L está em $\boldsymbol{NC}^d$ se L pode ser decidida por uma família de circuitos $\{C_n\}$ onde C_n tem tamanho limitado por um polinômio em n e profundidade $O(log^d n)$. A classe **NC** é definida como $\bigcup_{c\geq 1} \boldsymbol{NC}^c$.*

Definição 2.3.6. *A classe $\boldsymbol{AC}^i$ é definida de forma semelhante de $\boldsymbol{NC}^i$, exceto que não é limitado o número de bits de entrada das portas* OR *e* AND *do circuito. A classe **AC** é definida como $\bigcup_{i\geq 0} \boldsymbol{AC}^i$.*

Definição 2.3.7. *Uma linguagem encontra-se na classe **logDCFL** se pode ser decidida em espaço logarítmico e tempo polinomial por uma MT determinística adicionalmente equipada com uma pilha.*

Para ilustrar a dificuldade dessas classes, mostraremos a seguinte hierarquia de classes.

$$AC^0 \subsetneq NC^1 \subseteq logDCFL \subseteq AC^1 \subseteq AC^1(logDCFL) \subseteq AC^2 \subseteq P$$

Uma maior compreensão dessas classes podem ser obtidas em [Vollmer 1999].

3 MUDANÇA DE ONTOLOGIAS

O campo responsável pelo estudo da alteração de ontologias é chamado de Mudança de Ontologias. Primeiro, na seção 3.1, vamos discutir um pouco sobre ontologias e seu uso. Na seção 3.2 será apresentada uma visão geral da área, explicando e classificando as suas subáreas, segundo o tipo de problema que resolvem e suas motivações práticas. Em especial, o versionamento de ontologias, que será o foco do trabalho, será investigado com mais detalhes na seção 3.4.

3.1 Ontologias

O termo *ontologia* teve seu uso original em filosofia para denominar um ramo da metafísica que estuda a natureza da existência [Antoniou e Harmelen 2004]. Mais recentemente, essa palavra tem sido usada em Computação para falar de um tipo de representação. Uma definição bastante usada é a seguinte "uma ontologia é uma especificação formal e explícita de uma conceitualização compartilhada" [Gruber 1993]. Em outras palavras, uma ontologia fornece um vocabulário compartilhado para a representação formal de um domínio, especificando propriedades de objetos ou conceitos.

Para representar ontologias de forma precisa, são usadas linguagens formais construídas para esse propósito. Há uma grande gama de linguagens de representação de ontologias. Alguns exemplos são OWL [Motik et al. 2009] e DAML+OIL [McGuinness et al. 2002]. É comum usarmos, também, um tipo especial de linguagens formais para resgatar informações de ontologias chamadas *linguagens de consultas*. Um exemplo bastante conhecido é o da linguagem SPARQL, usada para consultar bancos de dados RDF, que geralmente representa instâncias do vocabulário definido em uma ontologia.

Muitas das linguagens de representação podem ser vistas como fragmentos da Lógica de Primeira Ordem. Em particular, as Lógicas de Descrição são muito usadas como um substrato formal para linguagens de representação de ontologias [Baader et al. 2003]. Por esse motivo, vamos usar para exemplos de ontologias a lógica $\mathcal{ALC}$ [Schmidt-Schauß e Smolka 1991] que apresentaremos a seguir.

3.1.1 A Lógica de Descrição $\mathcal{ALC}$

Na Lógica de Descrição $\mathcal{ALC}$ e na maioria dessas lógicas, trabalhamos com a ideia de conceitos e papéis. Eles representam os predicados unários e binários respectivamente. Assim, é possível denotar propriedades de um certo indivíduo e suas relações com outros indivíduos.

Na linguagem da $\mathcal{ALC}$, só é permitida a definição de novos conceitos. Os conceitos criados a partir de outros são chamados conceitos complexos. Para realizar essa tarefa, é necessário partir de alguns preexistentes, chamados de conceitos atômicos. Isso é feito por meio de construtores que serão mostrados na seguinte definição.

Definição 3.1.1 ($\mathcal{ALC}$-conceitos)**.** *Sejam N_C,N_R, respectivamente, conjuntos dos nomes de conceitos atômicos e papéis, os $\mathcal{ALC}$-conceitos é o menor conjunto definido indutivamente como segue:*

- $\top$ *é um $\mathcal{ALC}$-conceito;*
- $\bot$ *é um $\mathcal{ALC}$-conceito;*
- *se $C \in N_C$, C é um $\mathcal{ALC}$-conceito;*
- *se C é um $\mathcal{ALC}$-conceito, $\neg C$ é um $\mathcal{ALC}$-conceito;*
- *se $R \in N_R$ e C é um $\mathcal{ALC}$-conceito, $\exists R.C$ é um $\mathcal{ALC}$-conceito;*
- *se $R \in N_R$ e C é um $\mathcal{ALC}$-conceito, $\forall R.C$ é um $\mathcal{ALC}$-conceito;*
- *se C e D são $\mathcal{ALC}$-conceitos, $C \sqcap D$ é um $\mathcal{ALC}$-conceito;*

$\mathcal{ALC}$ permite relacionar conceitos através dos axiomas terminológicos definidos a seguir.

Definição 3.1.2. ***(Axiomas Terminológicos)***
A seguir, definiremos os axiomas terminológicos. Existem dois tipos:

- *Axiomas de inclusão: $C \sqsubseteq D$, com C e D sendo $\mathcal{ALC}$-conceitos;*
- *Axiomas de igualdade: $C \equiv D$, com C e D sendo $\mathcal{ALC}$-conceitos.*

Também é possível, em $\mathcal{ALC}$ representar propriedades de indivíduos, dizendo que um indivíduo pertence a um conceito ou que uma dupla de indivíduos pertence a um papel. A isso se dá o nome de Asserção de Conceitos e Papéis como definiremos a seguir.

Definição 3.1.3. ***(Asserção de Conceitos)***
"a" é chamado de instância de C, em que C é um conceito e "a" um indivíduo. Isso é representado da seguinte forma: $C(a)$ ou $a : C$.

Definição 3.1.4. ***(Asserção de Papéis)***
(a,b) é chamado de instância de R, em que R é um papel e a e b são indivíduos. Representa-se este fato da seguinte forma: $R(a,b)$ ou $\langle a,b\rangle : R$.

A partir dessas definições, diferenciamos *ABox* e *TBox*. O *ABox* é um conjunto de asserções de conceitos e papeis enquanto que o *TBox* é um conjunto de axiomas terminológicos. Para nós, nos exemplos de ontologias, não faremos essa distinção com clareza. Definiremos uma ontologia somente como a união dessas duas partes. Apesar disso, essa separação é bastante importante na literatura das lógicas de descrição.

A semântica da lógica $\mathcal{ALC}$ é definida considerando os conceitos subconjuntos do domínio, designado por $\Delta^{\mathcal{I}}$, e os papéis subconjuntos de $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. A interpretação dos construtores é dada segundo a seguinte definição.

Definição 3.1.5. *Dada uma interpretação $\mathcal{I} = \left\langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \right\rangle$ em que $\Delta^{\mathcal{I}}$ representa o domínio e $\cdot^{\mathcal{I}}$ representa uma função que mapeia cada conceito C em um conjunto $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ e cada papel R em um conjunto $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, os $\mathcal{ALC}$-conceitos são interpretados da seguinte forma:*

$$\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \varnothing \\
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \mid \textit{para todo } (a,b) \in R^{\mathcal{I}} \textit{ implica } b \in C^{\mathcal{I}}\} \\
(\exists R.\top)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \mid \textit{existe } (a,b) \in R^{\mathcal{I}}\}
\end{aligned}$$

As noções de satisfação para os axiomas terminológicos e para asserções de conceitos e papéis seguem abaixo:

Definição 3.1.6. ***(Satisfação dos axiomas terminológicos)***
Os axiomas terminológicos são satisfeitos por uma interpretação $\mathcal{I}$, ou seja,

- *$\mathcal{I} \vDash C \sqsubseteq D$ se e somente se $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$;*
- *$\mathcal{I} \vDash C \equiv D$ se e somente se $C^{\mathcal{I}} = D^{\mathcal{I}}$;*

Definição 3.1.7. ***(Satisfação de uma asserção de conceito)***
Uma asserção de conceito $C(a)$ é satisfeita por uma interpretação $\mathcal{I}$, ou seja, $\mathcal{I} \vDash C(a)$ se e somente se $a^{\mathcal{I}} \in C^{\mathcal{I}}$.

Definição 3.1.8. ***(Satisfação de uma asserção de papel)***
Uma asserção de papel $R(a,b)$ é satisfeita por uma interpretação $\mathcal{I}$, ou seja, $\mathcal{I} \vDash R(a,b)$ se e somente se $\langle a^{\mathcal{I}}, b^{\mathcal{I}} \rangle \in R^{\mathcal{I}}$.

3.2 Visão Geral da Área

Em [Flouris et al. 2008], são identificados 10 subcampos da área de mudança de ontologias. São eles: mapeamento de ontologias (do inglês *ontology mapping*), morfismos de ontologias (do inglês *ontology morphisms*), casamento de ontologias (do inglês *ontology matching*), articulação de ontologias (do inglês *ontology articulation*), tradução de ontologias (do inglês *ontology translation*), evolução de ontologias (do inglês *ontology evolution*), depuração de ontologias (do inglês *ontology debugging*), versionamento de ontologias (do inglês *ontology versioning*), integração de ontologias (do inglês *ontology integration*) e entrelaçamento de ontologias (do inglês *ontology merging*).

Podemos agrupar essas subáreas de acordo com a semelhança dos problemas de que tratam. Ainda em [Flouris et al. 2008], os primeiros 5 campos citados acima (mapeamento, morfismos, casamento, articulação e tradução de ontologias) são agrupados por tratarem de

resolução de heterogeneidades. O objetivo é lidar com diferenças quanto à terminologia, representação sintática, etc. De uma forma geral, os problemas tratados por esse grupo são resolvidos por meio de regras de tradução entre ontologias diferentes. O objetivo dessas regras, bem como sua representação, são especifidades de cada subcampo. É importante notar que as mudanças contidas nesse grupo são pré-requisitos para os outras que serão estudados, já que esses tomam por base que conflitos relacionados a, por exemplo, linguagem de representação, vocabulário e construções sintáticas estejam resolvidos.

Outro agrupamento que pode ser feito é entre evolução e depuração de ontologias. Enquanto na depuração de ontologias resolvemos erros de uma ontologia, restaurando propriedades como consistência e coerência, na evolução de ontologias modificamos a a ontologia segundo obtenção de novas informações à respeito do domínio, ou de mudanças no projeto da ontologia. Apesar dessas modificações não serem diretamente relacionadas ao versionamento, elas representam um passo importante. Por isso, discutiremos um pouco sobre evolução em 3.3.

Por último, o subcampo de versionamento de ontologias não tem nenhum outro subcampo com o qual pode ser agrupado. Esse será o problema central desse trabalho e trataremos desse problema com mais detalhes na seção 3.4.

3.3 Evolução de Ontologias

Em [Flouris e Plexousakis 2005], os autores definem essa subárea como "processo de modificar uma ontologia em resposta a uma mudança no domínio ou em sua conceitualização". Assim, dado um conjunto de requisitos de mudança, modifica-se a ontologia de forma a satisfazer os requisitos especificados. Esse processo gera uma versão que será administrada no Versionamento de Ontologias.

Para automatizar o processo de evolução, foi proposta a adaptação do formalismo já existente de mudança de crenças às ontologias [Flouris 2006]. Mudança de Crenças é a subárea da Representação do Conhecimento que trata de como um agente muda o que ele acredita de acordo com uma nova informação. Para isso, são criados operadores de mudança de crenças. Um vasto estudo foi feito para operadores para bases descritas em Lógica Proposicional. Ao automatizar a evolução com operadores de Mudança de Crenças, a interferência humana seria consideravelmente diminuída. Teríamos, também, uma noção de qualidade garantida pelas propriedades do operador de mudança de crenças. Tal operador deve atuar diretamente sobre as ontologias. Por essas serem representadas, em muitos casos, com lógicas mais expressivas como as Lógicas de Descrição, os operadores de Lógica Proposicional não podem ser diretamente utilizados. Muitos trabalhos tem sido feitos nessa área com operadores já propostos [Ribeiro et al. 2009, Liu et al. 2006, Qi, Liu e Bell 2006].

3.4 Versionamento de Ontologias

A *World Wide Web* é um ambiente bastante heterogêneo e pouco regulado. Isso pode ser exemplificado pela liberdade e facilidade que qualquer um tem para acessar e disponibilizar

conteúdo através dessa rede. O mesmo pode ser dito da criação e manutenção de ontologias. Diversas ontologias podem importar uma ontologia O assim como outras ontologias podem usar uma ontologia que importa O. Isso tudo acontece sem que os desenvolvedores de O tenham conhecimento. A modificação dessa ontologia pode prejudicar o funcionamento das aplicações que usam O diretamente, ou através de outra que a importa. Para evitar isso, são disponibilizadas várias versões de uma mesma ontologia. Assim, quando ontologias são modificadas, uma nova versões é criada ao mesmo tempo que é mantida uma versão sem a modificação, para evitar problemas acarretados por modificações.

Por outro lado, ao manter várias versões de uma ontologia, surgem alguns problemas. Esses são estudados no campo de Versionamento de Ontologias. Um dos problemas básicos é a identificação de versões. Esse problema consiste em identificar se uma modificação necessariamente consiste em outra versão. Qualquer modificação no arquivo da ontologia gera uma nova versão? Uma análise que leva em conta a semântica das versões pode mostrar que há modificações que não alteram os seus modelos. Entretanto, duas ontologias com os mesmos modelos são modificadas da mesma forma, ontologias diferentes podem ser obtidas. Assim, esse problema não é tão simples quanto parece. Abordagens para resolvê-lo podem ser encontradas em [Heflin, Hendler e Luke 1999] e [Klein et al. 2002].

Outro problema é prover acesso transparente a diversas versões de uma ontologia. Ao modificar uma ontologia, indivíduos, por exemplo, podem pertencer a uma versão v_1 e não pertencer a v_2. Assim qualquer consulta que referencie algum desses indivíduos deve ser redirecionada a v_1. Note que um indivíduo pode pertencer a vários conceitos, as definições dos conceitos a serem usadas na consulta devem ser da versão v_1 ou de v_2? Sempre desejamos ter informações mais novas possíveis, o que nos leva a v_2. Porém, os conceitos definidos nessa versão podem gerar contradições com os indivíduos de v_1. Esse problema não é trivial e está relacionado, intimamente, à análise de compatibilidade entre ontologias [Klein e Fensel 2001].

Listar as diferenças entre duas versões é outro problema de Versionamento de Ontologias. Abordagens que levam em conta as análises semânticas ou sintáticas entre as versões são formas de tentar resolver esse problema. Algumas abordagens fazem uso de metadados que descrevem as versões em questão. Algumas ferramentas foram desenvolvidas para listar tais diferenças, no entanto, muito ainda depende da intervenção humana [Noy e Musen 2002].

Em [Huang e Stuckenschmidt 2005], utiliza-se Lógica Temporal para raciocinar sobre versionamento de ontologias. Os autores proveem um formalismo para acessar diversas versões de uma ontologia e uma linguagem para raciocinar sobre essas versões. Nosso trabalho estenderá em vários sentidos este trabalho de Huang e Stuckenschmidt. No trabalho deles, não é feito nenhum estudo de complexidade da solução obtida, somente são abordadas versionamentos que ocorrem linearmente, ou seja, não há ramificações, e só é possível raciocinar sobre versões anteriores a uma dada versão.

4 LÓGICA MODAL

Nesse Capítulo, iremos introduzir as Lógicas Modais para posteriormente aplicar Lógicas Modais Temporais ao Versionamento de Ontologias. Nos concentraremos nas lógicas modais proposicionais, ou seja, lógicas proposicionais que são acrescidas de operadores modais. Faremos isso, pois essas lógicas tem semântica favorável para o tratamento dos problemas que serão atacados, como discutiremos no capítulo 5. Apresentaremos, também, um estudo das expressividades das lógicas apresentadas.

Na seção 4.1, definiremos alguns conceitos básicos e exemplos de lógicas modais simples. Posteriormente, na seção 4.2, veremos Lógicas Temporais e algumas extensões híbridas de Lógicas Temporais que serão de interesse para modelar o versionamento de ontologias no restante do trabalho. Finalmente, na seção 4.3, discutiremos o problema de *Model Checking*, analisando resultados de complexidade desse problema para as lógicas vistas no decorrer do capítulo.

4.1 Conceitos Básicos

Não há definição consensual de Lógica Modal pelos que estudam a área. Porém, podemos ressaltar algumas características comuns dessas lógicas. As Lógicas Modais permitem tratar de indivíduos por meio de estruturas relacionais, ou seja, estruturas que representam um conjunto e relações sobre esse conjunto. Entretanto, isso não é feito como na Lógica de Primeira ordem onde os elementos e suas relações são citados por elementos da linguagem. Para falar das relações, as Lógicas Modais tem operadores especiais, chamados de operadores modais. Em sua forma mais clássica, as Lógica Modais lidam com estruturas sob uma perspectiva local, ou seja, a partir de um elemento do domínio (também chamado de estado, ponto e vértice) expressamos somente características de estados relacionados a ele. Para expressar essa relação ou acessibilidade entre os elementos, são usados os operadores modais.

Mostraremos algumas características das Lógicas Modais Proposicionais por meio da mais simples delas, que veremos a seguir. A referência para todas as definições dessa seção é [Blackburn, Rijke e Venema 2002].

4.1.1 Sintaxe

A linguagem das Lógicas Modais Proposicionais, em sua forma geral, contém a Lógica Proposicional clássica, com a adição de operadores modais. Caso mais de um operador modal seja usado, essas são chamadas de multimodais.

Chamamos de Linguagem Modal Básica a linguagem que contém a lógica proposicional clássica e um único operador modal unário. A seguir, a definiremos com mais precisão.

Definição 4.1.1 (Linguagem Modal Básica)**.** *A Linguagem Modal Básica é definida usando um conjunto de símbolos proposicionais* Φ *e um operador modal* $\Diamond$ *('diamante'). As fórmulas bem*

formuladas ϕ da linguagem modal básica são das formas estabelecidas pela regra abaixo, em que p é um elemento de Φ.

$$\phi ::= p \mid \bot \mid \neg\phi \mid (\phi \vee \phi) \mid \Diamond\phi$$

Podemos estender a definição da Linguagem Modal Básica de forma a obter linguagens mais gerais. Para isso, define-se o que é um tipo de similaridade, responsável por descrever os operadores de uma linguagem.

Definição 4.1.2 (Tipo de Similaridade Modal)**.** *Um tipo de similaridade modal é um par $\tau = (O,\rho)$, em que O é um conjunto não vazio e ρ é uma função $O \to \mathbb{N}$. Os elementos de O são chamados de operadores modais; usamos $\triangle_0, \triangle_1, \ldots$ para denotar os elementos de O. A função ρ associa a cada operador $\triangle \in O$ uma aridade finita, indicando o número de argumentos de $\triangle$.*

Com base em um tipo de similaridade, definimos uma Linguagem Modal como segue.

Definição 4.1.3 (Linguagem de uma Linguagem Modal)**.** *Uma linguagem modal $ML(\tau,\Phi)$ é construída usando um tipo de similaridade modal $\tau = (O,\rho)$ e um conjunto de letras proposicionais Φ. O conjunto $Form(\tau,\Phi)$ de fórmulas modais sobre τ e Φ é dado pela regra abaixo em que $p \in \Phi$ e $\triangle \in O$.*

$$\phi ::= p \mid \bot \mid \neg\phi \mid (\phi \vee \phi) \mid \triangle(\underbrace{\phi, \ldots, \phi}_{\rho(\triangle)})$$

4.1.2 Semântica

4.1.2.1 *Frames*

A interpretação das fórmulas das Lógicas Modais é feita sobre estruturas relacionais, ou seja, estruturas com um domínio e relações sobre esse domínio. Representamos essas estruturas por meio do que chamamos de *frames*, como definiremos a seguir.

Definição 4.1.4 (*Frame*)**.** *Um* frame *para Lógica Modal Básica é um par $\mathfrak{F} = (W,R)$ tal que*

1. *W é um conjunto não vazio,*
2. *R é uma relação binária em W.*

Em W, estarão os estados de um *frame*. Já em R, serão indicados como esses estados se relacionam. A relação R será responsável por definir acessibilidade entre os estados dos operadores modais. Por isso, ela é chamada de relação de acessibilidade.

A definição acima de *frame* não suporta mais de um operador modal. Por isso, a seguir, a definição de *frame* é generalizada para permitir operadores modais de acordo com um tipo de similaridade modal.

Definição 4.1.5 (*Frame* generalizado)**.** *Seja τ um tipo de similaridade modal. Um τ-*frame *é uma tupla $\mathfrak{F}$ consistindo dos seguintes ingredientes*

1. *W é um conjunto não vazio,*
2. *para cada $n \geq 0$, e cada operador modal n-ário no tipo de similaridade τ, uma relação $(n+1)$-ária $R_\triangle$.*

Podemos classificar frames segundo as propriedades da sua relação de acessibilidade. Um tipo de frame que será usado no decorrer do trabalho é a Árvore. Vejamos sua definição.

Definição 4.1.6 (Árvore)**.** *Uma árvore $\mathcal{T}$ é uma estrutura relacional (T, S) em que:*

- *T, o conjunto de nós, contém um único $r \in T$ (chamado de raiz) tal que para todo $t \in T$, S^*rt, onde S^* é o fecho transitivo reflexivo da relação S.*
- *Todo elemento de T distinto de r tem um único S-predecessor, ou seja, para todo $t \neq r$ existe um único $t' \in T$ tal que $St't$.*
- *S é acíclico, ou seja, para todo t, não temos S^+tt, onde S^+ representa o fecho transitivo de S. Isto é, não podemos alcançar t através de t.*

4.1.2.2 Modelos

Com os *frames*, somos capazes de verificar como estados diferentes se relacionam. Porém, ainda não temos como verificar propriedades de um dado estado. Para isso, a noção de valoração será usada. A partir de um *frame* e uma valoração sobre os estados do frame, temos um modelo. Vejamos a definição de um modelo para a Linguagem Modal Básica.

Definição 4.1.7 (Modelo para a Lógica Modal Básica)**.** *Um modelo para Lógica Modal Básica é um par $\mathfrak{M} = (\mathfrak{F}, V)$ em que $\mathfrak{F} = (W, R)$ é um* frame *para a Lógica Modal Básica e V é uma função que assinala a cada letra proposicional $p \in \Phi$ um elemento de $\mathcal{P}(W)$, onde $\mathcal{P}(W)$ representa o conjunto das partes de W. A função V é chamada de valoração.*

Da mesma forma, a definição anterior pode ser generalizada para um dado tipo de similaridade τ. O resultado é a definição de modelo para uma linguagem modal arbitrária, segundo seu tipo de similaridade.

Definição 4.1.8 (Modelo para uma Lógica Modal)**.** *Seja τ um tipo de similaridade modal e $\mathfrak{F} = (W, R_\triangle)_{\triangle \in \tau}$ um τ-*frame*. Um τ-modelo é um par $\mathfrak{M} = (\mathfrak{F}, V)$, em que V é uma valoração com domínio em Φ e imagem em $\mathcal{P}(W)$, onde $\mathcal{P}(W)$ representa o conjunto das partes de W.*

Por último, definiremos a noção de satisfação modal para a lógica modal básica.

Definição 4.1.9 (Satisfação para a Lógica Modal Básica)**.** *Seja w um estado em um modelo $\mathfrak{M} = (W, R, V)$. Definimos indutivamente a noção de uma fórmula ϕ ser satisfeita em um modelo $\mathfrak{M}$ em w da seguinte forma:*

$$\begin{aligned}
\mathfrak{M}, w \Vdash p \quad & sse \quad w \in V(p), \text{ com } p \in \Phi \\
\mathfrak{M}, w \Vdash \bot \quad & \quad\;\; nunca \\
\mathfrak{M}, w \Vdash \neg\phi \quad & sse \quad \text{não } \mathfrak{M}, w \Vdash \phi \\
\mathfrak{M}, w \Vdash (\phi \vee \psi) \quad & sse \quad \mathfrak{M}, w \Vdash \phi \text{ ou } \mathfrak{M}, w \Vdash \psi \\
\mathfrak{M}, w \Vdash \Diamond\phi \quad & sse \quad \text{para algum } v \in W \text{ com } Rwv \text{ temos } \mathfrak{M}, v \Vdash \phi
\end{aligned}$$

Para generalizar a definição de satisfação para operadores modais de aridades arbitrárias, basta modificar o caso modal da definição.

Definição 4.1.10 (Satisfação para uma Lógica Modal)**.** *Seja w um estado em um modelo $\mathfrak{M} = (W, R, V)$. Definimos indutivamente a noção de uma fórmula ϕ ser satisfeita em um modelo $\mathfrak{M}$ em w da seguinte forma:*

$$\begin{aligned}
\mathfrak{M}, w \Vdash \triangle(\phi_1, \ldots, \phi_n) \text{ e } n > 0 \quad & sse \quad \text{para algum } v_1, \ldots, v_n \in W \text{ com } Rwv_1 \cdots v_n \\
& \qquad\;\; \text{temos para cada } i\text{ , } \mathfrak{M}, v_i \Vdash \phi_i \\
\mathfrak{M}, w \Vdash \triangle \quad & sse \quad w \in R_{\triangle}
\end{aligned}$$

Outras noções semânticas importantes são definidas para as Lógicas Modais. Exemplos importantes são as definições de verdade global(do inglês *globally true*), validade e consequência lógica. Outra definição muito importante é a bissimulação. Ela mostra condições de equivalência de modelos com respeito a noção de satisfação de uma determinada linguagem modal. Apesar da sua importância, não mostraremos essas definições, pois não as utilizaremos nesse trabalho. Para uma referência, consulte [Blackburn, Rijke e Venema 2002].

A definição de satisfação está intimamente ligada com *Model Checking*. Posteriormente, na seção 4.3, apresentaremos a definição formal do problema, mas como esse é um assunto importante para nosso trabalho, daremos uma intuição. Para uma fórmula e um modelo de uma lógica modal, desejamos saber em que estados essa fórmula é satisfeita. Para fazermos essa checagem, devemos utilizar a definição de satisfação da lógica em questão. Essa noção é aplicável para descrever o comportamento de sistemas. As Lógicas Temporais, que apresentaremos a seguir, são bastante utilizadas para isso.

4.2 Lógicas Temporais

O início do que hoje chama-se de Lógica Temporal teve início com as ideias de Prior nos anos 50 [Prior 1957]. Ele então as chamava de Lógicas Tensas que eram extensões da Lógica Proposicional Clássica com a adição de operadores modais para falar de noções temporais. A primeira dessas lógicas foi o que hoje chamamos de Lógica Temporal Básica e será vista com mais detalhes em 4.2.1.

A ideia fundamental dessas lógicas era analisar sistemas dinâmicos, ou seja, sistemas cuja configuração muda com o passar do tempo. Para isso seria necessário analisar a evolução ou variação dos estados do sistema. De forma geral, elas fornecem meios de expressar propriedades de estados e de uma sucessão deles.

As Lógicas Temporais tem sido bastante aplicadas em Computação na área de verificação formal de sistemas [Clarke, Grumberg e Peled 1999]. A verificação formal requer a representação dos estados que o sistema pode assumir, juntamente com as transições entre esses estados. Propriedades desejáveis e indesejáveis devem ser expressas por fórmulas dessas lógicas e, posteriormente, testamos se, nos estados do sistema, as fórmulas de propriedades desejáveis são preservadas e se as fórmulas de propriedades indesejáveis nunca são satisfeitas. Assim, pode-se testar um sistema e, possivelmente, identificar erros sem a necessidade de colocar o sistema real em funcionamento.

Essas características das Lógicas Temporais podem ser aproveitadas para especificação do versionamento de uma ontologia. No Capítulo 5, mostraremos como modificações das Lógicas Temporais que veremos podem ser usadas para especificar propriedades de versões de uma ontologia.

Cada lógica temporal adota uma forma de analisar o tempo. Pode-se definir dois grandes grupos de lógicas temporais, as que tratam de tempo linear e as que tratam de tempo ramificado. Considerando tempo linear, para cada estado, há no máximo um estado seguinte. Segundo essa visão, o futuro de um dado estado é determinado, não sujeito a mudanças. Analisando graficamente, temos uma cadeia de estados. Assim, em cada ponto da cadeia sabe-se qual é o próximo estado.

A visão ramificada de tempo, ao contrário da linear, permite considerar vários possíveis futuros de um determinado estado. Assim, dependendo das escolhas não-determinísticas de um agente ou da ocorrência ou não de um evento, diferentes situações podem ser obtidas a partir de uma mesma situação inicial. A representação gráfica que antes era de uma cadeia passa a ser um grafo arbitrário.

As lógicas que estudaremos nessa seção podem ter suas linguagens aplicadas tanto à visão linear quanto à ramificada, dependendo da sua semântica. Algumas das lógicas que serão apresentadas, nomeadamente LTL,CTL e CTL*, são exemplos clássicos de lógicas temporais. Muito foi feito nesse campo desde a sua criação, como estudo de lógicas de intervalos [Goranko, Montanari e Sciavicco 2004] e lógicas com quantificação em proposições [Sistla, Vardi e Wolper 1987]. Porém, as lógicas apresentadas oferecem expressividade suficiente para tratar o problema, enquanto tem uma base sólida de resultados, dos quais tomaremos proveito em 4.3.

Para manter simples a apresentação dessas lógicas, todas elas, com exceção da Lógica Temporal Básica, serão definidas neste capítulo sem operadores que tratam de passado. Em 4.2.5, esses operadores modais serão estudados.

4.2.1 Lógica Temporal Básica

A linguagem dessa lógica é bastante simples, apresentando um operador para falar de futuro e outro para passado. Ela pode ser apresentada no formato da definição 4.1.3.

Definição 4.2.1 (Linguagem Temporal Básica)**.** *A Linguagem Temporal Básica é construída usando o conjunto de operadores* $O = \{\langle F\rangle, \langle P\rangle\}$*, ambos unários.*

Indicamos por $\langle\rangle$ que o operador tem um comportamento semelhante ao operador $\Diamond$. Para simplificar a notação, escreveremos, como é habitual na área, F para denotar $\langle F\rangle$.

Note que, por estarmos tratando de uma linguagem, não há restrição quanto à acessibilidade dos operadores F e P, aceitando relações binárias arbitrárias, apesar da interpretação pretendida de $F\phi$ ser *'ϕ será satisfeita em algum momento'* e de $P\phi$ ser *'ϕ foi satisfeita em algum momento'*.

Como essa lógica é bastantes simples, não apresenta expressividade suficiente para especificar fórmulas que indiquem a repetição de uma propriedade ou a impossibilidade de se chegar a um estado que tenha uma propriedade. Esse tipo de fórmula é muito usada sistemas em que há concorrência por recurso. Por exemplo 'o sistema nunca vai apresentar um estado de *deadlock*'. A lógica que apresentaremos a seguir é mais expressiva, pois permite expressar propriedades de caminhos.

4.2.2 CTL*

A lógica CTL* (do inglês *Computation Tree Logic*) foi desenvolvida em meados dos anos 80 [Clarke, Emerson e Sistla 1986]. Essa lógica é uma extensão da famosa lógica CTL, que veremos em 4.2.3. CTL* é bastante expressiva e largamente estudada no âmbito de lógicas temporais. Como seu nome leva a entender, com essa lógica podemos raciocinar sobre propriedades de árvores de computação. Entretanto, vale ressaltar que seus *frames* não são, necessariamente, em forma de árvore. As árvores de computação são uma abstração da sequência de estados percorridos para validar uma fórmula. Essa lógica foi construída modelar o comportamento de de programas, principalmente concorrentes. Por isso, a árvore obtida descreveria os passos da computação realizada.

Vejamos o exemplo ilustrado na figura 4.1 de um *frame* e sua árvore de computação correspondente. Percorrendo o *frame* representado pelo desenho 4.1a a partir de S_0 para testar uma fórmula que requer visita de todos os caminhos do frame, obtemos a árvore de computação do desenho 4.1b.

Note que, apesar do *frame* ser finito, devido aos ciclos, a árvore de computação gerada é infinita. Isso não significa que, para avaliar um fórmula a partir de S_0, devemos visitar todos estados subsequentes da árvore infinitamente. Caso fizéssemos um programa que agisse dessa forma, ele não pararia.

Em CTL*, é possível representar propriedades de estados e caminhos da árvore de computação de um *frame*. Vejamos como sua sintaxe permite fórmulas para isso. Dividiremos

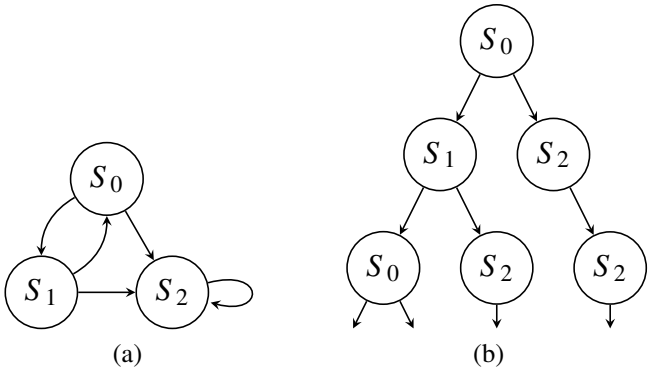

Figura 4.1: Exemplo de árvore de computação

as fórmulas em dois tipos: estado e caminho. Quando estudarmos a semântica dessa lógica, essa distinção será deixada mais clara. Nessa definição as fórmulas de estado serão representadas pelo símbolo ϕ e as de caminho por ψ.

Definição 4.2.2 (Linguagem de CTL*)**.** *Seja um conjunto de letras proposicionais* Φ*. O conjunto de fórmulas de CTL* com* Φ *é dado pela regra abaixo em que* $p \in \Phi$*.*

$$\begin{aligned} \phi &::= p \mid \bot \mid \neg\phi \mid (\phi \vee \phi) \mid E\psi \\ \psi &::= \phi \mid \neg\psi \mid (\psi \vee \psi) \mid X\psi \mid (\psi U \psi) \end{aligned}$$

Notemos que toda fórmula de estado é também uma fórmula de caminho e que os conectivos lógicos proposicionais são aplicados a ambos os tipos. Esses detalhes serão refletidos diretamente na semântica de CTL*.

Para definir a satisfação para fórmulas de CTL*, usaremos a noção de caminho em um *frame* que é usada diretamente em CTL*.

Definição 4.2.3 (Caminho em um *frame*)**.** *Um caminho em um frame* $\mathfrak{F}$ *é uma sequência infinita de estados* $\pi = w_0 w_1 w_2 \ldots$ *tal que, para todo* $i \leq 0$*, temos* $Rw_i w_{i+1}$*. Denotamos por* π^i *o sufixo de* π *começando em* w_i*.*

A definição de caminho é melhor entendida quando pensamos na árvore de computação de uma fórmula. Os caminhos da definição anterior podem ser pensados como ramos da árvore de computação. Vejamos então a definição de satisfação em CTL*.

Definição 4.2.4 (Satisfação para CTL*)**.** *Sejam w um estado em um modelo* $\mathfrak{M} = (W, R, V)$*,* ϕ_1 *e* ϕ_2 *fórmulas de estado e* ψ_1 *e* ψ_2 *fórmulas de caminho. Definimos indutivamente a noção de*

uma fórmula de CTL ser satisfeita em um modelo $\mathfrak{M}$ em w da seguinte forma:*

$$
\begin{array}{rll}
\mathfrak{M}, w \Vdash p & sse & w \in V(p), \text{ com } p \in \Phi \\
\mathfrak{M}, w \Vdash \bot & & \text{nunca} \\
\mathfrak{M}, w \Vdash \neg\phi_1 & sse & \text{não } \mathfrak{M}, w \Vdash \phi_1 \\
\mathfrak{M}, w \Vdash (\phi_1 \vee \phi_2) & sse & \mathfrak{M}, w \Vdash \phi_1 \text{ ou } \mathfrak{M}, w \Vdash \phi_2 \\
\mathfrak{M}, w \Vdash E\phi_1 & sse & \text{existe } \pi = w\pi^1 \text{ tal que } \mathfrak{M}, \pi \Vdash \phi_1 \\
\mathfrak{M}, \pi \Vdash \phi_1 & sse & \pi = w\pi^1 \text{ e } \mathfrak{M}, w \Vdash \phi_1 \\
\mathfrak{M}, \pi \Vdash \neg\psi_1 & sse & \text{não } \mathfrak{M}, \pi \Vdash \psi_1 \\
\mathfrak{M}, \pi \Vdash (\psi_1 \vee \psi_2) & sse & \mathfrak{M}, \pi \Vdash \psi_1 \text{ ou } \mathfrak{M}, \pi \Vdash \psi_2 \\
\mathfrak{M}, \pi \Vdash X\psi_1 & sse & \mathfrak{M}, \pi^1 \Vdash \psi_1 \\
\mathfrak{M}, \pi \Vdash (\psi_1 U \psi_2) & sse & \text{existe } k \geq 0 \text{ tal que } \mathfrak{M}, \pi^k \Vdash \psi_2 \text{ e, para} \\
 & & \text{todo } 0 \leq j \leq k, \mathfrak{M}, \pi^j \Vdash \psi_1
\end{array}
$$

Alguns operadores podem ser obtidos a partir dos operadores básicos apresentados de acordo com a semântica apresentada. Consideramos esses operadores como abreviações de fórmulas de CTL*. São eles:

$$
\begin{aligned}
\top &= \neg\bot \\
A\psi &= \neg E\neg\psi \\
F\psi &= (\top U \psi) \\
G\psi &= \neg F\neg\psi \\
\phi R\psi &= \neg(\neg\psi U \neg\phi)
\end{aligned}
$$

Para melhor entendimento, apresentamos também a semântica desses operadores, derivada da satisfação dos operadores de CTL* já apresentados.

$$
\begin{array}{rll}
\mathfrak{M}, w \Vdash A\phi_1 & sse & \text{para todo } \pi = w\pi^1, \mathfrak{M}, \pi \Vdash \phi_1 \\
\mathfrak{M}, \pi \Vdash F\psi_1 & sse & \text{existe } k \geq 0 \text{ tal que } \mathfrak{M}, \pi^k \Vdash \psi_1 \\
\mathfrak{M}, \pi \Vdash G\psi_1 & sse & \text{para todo } k \geq 0, \mathfrak{M}, \pi^k \Vdash \psi_1 \\
\mathfrak{M}, \pi \Vdash \psi_1 R \psi_2 & sse & \text{para todo } j \geq 0, \text{ se para todo } i < j \text{ não } \mathfrak{M}, \pi^i \Vdash \psi_1 \text{ então} \\
 & & \mathfrak{M}, \pi^j \Vdash \psi_2
\end{array}
$$

Podemos traduzir algumas fórmulas de CTL em linguagem natural, como

- AFp, "em todos os caminhos, no futuro vale p"
- $EF(Ap \wedge Eq)$, "existe um caminho que no futuro, em todos os caminhos vale p e existe um caminho em que vale q"

4.2.3 CTL

A lógica CTL é uma restrição sintática de CTL* definida por Emerson e Clarke em [Emerson e Clarke 1982]. Nessa lógica, ao contrário de CTL*, não é possível aplicar os operadores X e U sem um operador E imediatamente anterior. Assim, fórmulas como $EXXp$ e $AXpUq$ não são fórmulas de CTL. Outra mudança é a adição do operador G à linguagem, já que, devido à restrição sintática anterior, não é possível obtê-lo a partir de outros operadores como em CTL*.

Da mesma forma que em CTL*, ϕ representará as fórmulas de estado e ψ as de caminho.

Definição 4.2.5 (Linguagem de CTL)**.** *Seja um conjunto de letras proposicionais* Φ*. O conjunto de fórmulas de CTL com base em* Φ *é dado pela regra abaixo em que* $p \in \Phi$*.*

$$\begin{aligned} \phi &::= p \mid \bot \mid (\neg\phi) \mid (\phi \vee \phi) \mid E\psi \\ \psi &::= X\phi \mid (\phi U \phi) \mid G\phi \end{aligned}$$

A definição da semântica é a mesma de CTL*. Devido às mudanças na linguagem, as abreviações de CTL* não podem ser obtidas da mesma forma em CTL. Vejamos a seguir, como algumas abreviações podem ser obtidas a partir da sintaxe de CTL.

$$\begin{aligned} AX\phi &= \neg EX\neg\phi \\ EF\phi &= E(\top U\phi) \\ AF\phi &= \neg EG\neg\phi \\ AG\phi &= \neg EF\neg\phi \\ A(\phi_1 U\phi_2) &= \neg E(\neg\phi_2 U(\neg\phi_1 \wedge \neg\phi_2)) \wedge \neg EG\neg\phi_2 \\ A(\phi_1 R\phi_2) &= E(\neg\phi_1 U\neg\phi_2) \\ E(\phi_1 R\phi_2) &= A(\neg\phi_1 U\neg\phi_2) \end{aligned}$$

Apesar de CTL* apresentar um ganho de expressividade com relação à CTL, CTL apresenta uma complexidade para resolução o problema de *Model Checking* bem menor, tornando-a mais apropriada para aplicações que requerem eficiência. Estudaremos detalhadamente em 4.3 a importância desse problema e as complexidades para diversas lógicas.

4.2.4 LTL

A Lógica LTL (*Linear Time Logic*) foi proposta por Pnueli [Pnueli 1977]. Essa lógica foi idealizada para especificação formal da execução de programas. Apesar de seu nome citar '*tempo linear*', ela também lida com ramificações, mas de uma forma diferente de CTL e CTL*. Comentaremos mais sobre ramificações após a apresentação da sintaxe de suas fórmulas.

Mesmo tendo sido proposta mais cedo que CTL e CTL*, ela pode ser vista como uma restrição sintática de CTL*. Da mesma forma que CTL* e ao contrário de CTL, a sintaxe de LTL permite que mais de um operador de caminho seja aplicado em sequência sem necessidade quantificador de caminho. Apesar dessa semelhança, LTL difere de CTL*, pois nem toda fórmula de estado é uma fórmula caminho. Somente os símbolos proposicionais e *bottom* ($\bot$) são em fórmulas de caminho. Outra peculiaridade de LTL é que não é possível usar o quantificador de caminho E. Isso impacta diretamente em como LTL lida com ramificações.

Novamente, assim como em CTL e CTL*, ϕ representará as fórmulas de estado e ψ as de caminho. Apresentaremos a definição de LTL de acordo com [Clarke, Grumberg e Peled 1999], onde fica claro que LTL é restrição de CTL*.

Definição 4.2.6 (Linguagem de LTL como restrição de CTL*)**.** *Seja um conjunto de letras proposicionais* Φ*. O conjunto de fórmulas de LTL com base em* Φ *é dado pela regra* ϕ *abaixo em que* $p \in \Phi$*.*

$$\begin{aligned} \phi &::= A\psi \mid P \\ \psi &::= P \mid \neg\psi \mid (\psi \vee \psi) \mid (\psi U \psi) \mid X\psi \\ P &:= p \mid \bot \end{aligned}$$

Da mesma forma que CTL, a semântica de LTL é a mesma de CTL*. Devido ao formato das fórmulas de LTL, temos uma grande diferença prática no que diz respeito ao tratamento dos *frames*. Como só temos fórmulas iniciadas pelo operador A, para que uma fórmula $A\psi$ seja satisfeita em um estado w, ψ deve ser satisfeita em todos os caminhos iniciados em w. Como os operadores A e E não ocorrem em ψ, não é possível quantificar propriedades de outros caminhos. Então, as sub-fórmulas de ψ só expressam propriedades dos segmentos do caminho em que ψ está sendo interpretada. Logo, em LTL, *frames* que não tem formato de cadeia tem seus caminhos analisados um a um segundo a propriedade especificada em ψ. Por isso a lógica é dita de tempo linear.

As expressividades de CTL*, CTL e LTL, já foram bastante estudadas, devido à importância dessas lógicas para o campo de Verificação Formal [Manna e Pnueli 1995, Clarke, Grumberg e Peled 1999]. Assim, temos que as expressividades de CTL e LTL são incomparáveis, ou seja, existem propriedades que são expressadas em CTL que não podem ser expressadas em LTL e vice-versa. Porém, existem fórmulas comuns a essas duas lógicas. Um importante resultado que relaciona fórmulas de CTL e LTL foi obtido por Clarke e Draghicescu em [Clarke e Draghicescu 1989], como veremos a seguir.

Teorema 4.2.1 ([Clarke e Draghicescu 1989])**.** *Seja* ψ *a fórmula de LTL obtida ao retirarmos*

todos os quantificadores de caminho de uma fórmula de CTL φ e colocarmos um quantificador A no início. Podemos expressar φ em LTL se e somente se ψ é equivalente a φ.

O teorema 4.2.1 fornece uma forma de construir uma fórmula de LTL equivalente a uma fórmula de CTL, caso essa exista.

CTL*, por sua vez, devido à sua sintaxe menos restrita, apresenta expressividade estritamente maior que CTL e LTL. Isso pode ser facilmente exemplificado se tomarmos duas fórmulas φ e ψ de CTL e LTL, respectivamente, as quais φ não tem equivalente em LTL e ψ não tem equivalente em CTL. A partir dessas fórmulas, construímos a fórmula $\varphi \wedge \psi$, que é uma fórmula de CTL*. Essa fórmula não tem equivalente em LTL nem em CTL.

A seguir, na figura 4.2 mostraremos o que foi explicado anteriormente, ilustrando alguns exemplos de fórmulas.

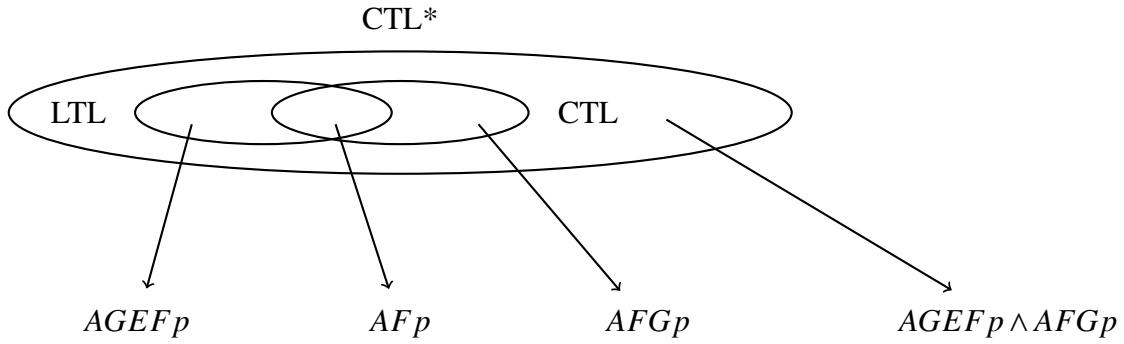

Figura 4.2: Expressividade de CTL*, CTL e LTL

Examinaremos um pouco mais da expressividade dessas lógicas em 5.4.

4.2.5 Operadores de Passado

As lógicas temporais podem ser estendidas com operadores que designam estados anteriores de um dado estado. Para LTL e variações dessa lógica, não há aumento de expressividade ao acrescentar esses operadores [Gabbay et al. 1980, Gabbay 1987]. Isso é provado por meio de traduções dos operadores de passado para fórmulas dessas linguagens temporais sem eles. Apesar disso, ao usar operadores de passado, é possível diminuir exponencialmente o tamanho de uma fórmula, obtendo fórmulas mais sucintas [Laroussinie, Markey e Schnoebelen 2002]. O tamanho da fórmula influencia o cálculo de complexidade de problemas que a tem como entrada, por exemplo o problema de *model checking*.

As lógicas LTL e CTL podem ser estendidas com um operador de passado, obtendo PLTL [Gabbay et al. 1980] e PCTL [Kupferman e Pnueli 1995]. Para lidar com a ideia que o passado de um dado estado é único, os frames usados são em formato de árvore como definido em 4.1.6. Os operadores dessa lógica são os mesmos das lógicas originais, incluindo Y para referir-se ao estado anterior e S para referir-se à modalidade "desde". O operador S é análogo do passado ao operador U.

Definição 4.2.7 (Linguagem de PLTL)**.** *Sejam ϕ_{LTL} as fórmulas de estado de LTL e ψ_{LTL} as fórmulas de estado de LTL.*

$$\begin{aligned} \phi &::= \phi_{LTL} \mid Y\phi \mid \phi S \phi \\ \psi &::= \psi_{LTL} \end{aligned}$$

Definição 4.2.8 (Linguagem de PCTL)**.** *Sejam ϕ_{CTL} as fórmulas de estado de CTL e ψ_{CTL} as fórmulas de estado de CTL.*

$$\begin{aligned} \phi &::= \phi_{CTL} \mid Y\phi \mid \phi S \phi \\ \psi &::= \psi_{CTL} \end{aligned}$$

Os operadores Y e S apresentam a mesma semântica tanto em PLTL e PCTL. Vejamos a satisfação desses novos operadores nas duas lógicas.

Definição 4.2.9 (Semântica de $\boldsymbol{Y}$ e $\boldsymbol{S}$)**.** *Seja $\mathfrak{M}$ um modelo baseado em um frame em formato de árvore$\mathcal{F} = (W, R)$ e $w \in W$*

$$\begin{aligned} \mathfrak{M}, w \Vdash Y\phi \quad & sse \quad (w', w) \in R \text{ e } \mathfrak{M}, w' \Vdash \phi \\ \mathfrak{M}, w \Vdash \phi S \psi \quad & sse \quad \text{existe um ancestral } w' \text{ de } w, \text{ tal que } \mathfrak{M}, w' \Vdash \psi \\ & \quad\; \text{e para qualquer estado } v \text{ entre } w' \text{ e } w, \text{ temos } \mathfrak{M}, v \Vdash \phi \end{aligned}$$

Apesar de PLTL não ser mais expressiva que LTL, o mesmo não vale para PCTL e CTL. A lógica PCTL é estritamente mais expressiva que CTL [Laroussinie e Schnoebelen 1995].

4.2.6 Caminhos Finitos

As lógicas temporais apresentam grande utilidade no campo de especificação formal do comportamento de programas. Pelo fato de termos definido as lógicas LTL, CTL e CTL* sobre caminhos infinitos, a aplicação dessas lógicas não é adequada a *frames* em que temos um limite para o tamanho dos caminhos e com isso garantia de finitude. Para lidar com isso, mostraremos uma possível semântica para caminhos finitos.

A única modificação diz respeito ao operador temporal X. Quando tratamos de frames finitos, $\mathfrak{M}, \pi \Vdash X\phi$ não está bem definida quando $\pi = \{u\}$, ou seja, quando estamos no último estado do caminho.

Para lidar com esse problema, Manna e Pnueli defendem que o operador de próximo estado X deva substituído por dois operadores: próximo fraco $\overline{X}$ e $\underline{X}$ dito forte em [Manna e Pnueli 1995]. Como veremos a seguir na definição da semântica de cada um, eles concordam com o operador X quando há um próximo estado para um determinado estado em um caminho, mas diferem entre si e de X quando não há. Ao usar esses operadores, é possível detectar o fim de um caminho.

Definição 4.2.10 (Semântica de $\underline{X}$ e $\overline{X}$)**.**

$$\mathfrak{M}, \pi \Vdash \overline{X}\phi \quad sse \quad \pi^1 = \varepsilon \; ou \; \mathfrak{M}, \pi^1 \Vdash \phi, \; com \; \pi^1 \neq \varepsilon$$
$$\mathfrak{M}, \pi \Vdash \underline{X}\phi \quad sse \quad \pi^1 \neq \varepsilon \; ou \; \mathfrak{M}, \pi^1 \Vdash \phi, \; com \; \pi^1 \neq \varepsilon$$

Essa semântica não é a única para tratar de frames finitos. No trabalho [Bauer, Leucker e Schallhart 2007], é apresentada uma semântica trivalorada dispensando os dois operadores de próximo. O terceiro valor indica a ausência de um próximo estado em um caminho. Já em [Bauer, Leucker e Schallhart 2009], os dois operadores são preservados e são adicionados mais dois valores, resultando em uma semântica de quatro valores. Outras abordagens são usadas em [Morgenstern, Gesell e Schneider 2012] onde são definidas várias semânticas específicas para uma hierarquia de lógicas diferentes. Apesar de promissoras, as abordagens desses trabalhos são bastante variadas, não havendo, ainda, consenso da melhor semântica. Por isso, no decorrer do trabalho, usaremos a semântica definida acima que é a mais conhecida.

4.2.7 Operadores de Lógica Híbrida

Pelo fato das lógicas modais, de forma geral, serem menos expressivas que a Lógica de Primeira Ordem, estudou-se como estender a expressividade dessas lógicas, tentando manter a propriedade de decidibilidade. Obteve-se, então, as lógicas chamadas de Lógicas Híbridas. Elas são denominadas dessa forma, pois tem por base lógicas modais, mas com operadores que permitem indicar explicitamente elementos do domínio das estruturas relacionais nas quais as fórmulas são interpretadas. Assim, compartilham características das Lógicas Modais e da Lógica de Primeira ordem.

Estenderemos, então, a sintaxe da lógica modal básica, adicionando um conjunto especial de símbolos proposicionais que serão usados para denominar os elementos do domínio. Esses símbolos proposicionais são chamados de nominais. Para uma noção geral de Lógica Híbrida, veja [Blackburn, Rijke e Venema 2002, Blackburn 2000].

Definição 4.2.11 (Linguagem da Lógica Híbrida)**.** *Sejam um conjunto de símbolos proposicionais Φ, um conjunto de nominais Ω disjuntos e um operador modal $\Diamond$ ('diamante'). As fórmulas ϕ da linguagem da Modal Básica com com nominais e o operador @ são como estabelecidas pela regra abaixo, em que $p \in \Phi$, $i \in \Omega$.*

$$\phi ::= i \mid p \mid \bot \mid \neg\phi \mid (\phi \vee \phi) \mid \Diamond\phi \mid @_i\phi$$

Definição 4.2.12 (Modelos híbridos e satisfação em Lógica Híbrida)**.** *Um modelo híbrido $\mathfrak{M} = (\mathfrak{F}, V)$ é composto por um* frame *$\mathfrak{F}$ e uma valoração V. Porém, devido ao acréscimo dos nominais na linguagem, a valoração deve mapear os símbolos proposicionais, bem como os nominais. Uma valoração híbrida é uma função que leva um elemento de $\Phi \cup \Omega$ em um elemento de $\mathcal{P}(W)$, sendo W o conjunto de estados de $\mathfrak{F}$. Para qualquer elemento $j \in \Omega$, $V(j)$ é um conjunto unitário, cujo único elemento é chamado denotação do nominal j. Quando $j \in \Phi$, $V(j)$ indica os*

estados em que o símbolo j é verdade, como usual em Lógica Modal. A satisfação das fórmulas exclusivas da Lógica Híbrida são como segue:

$$\mathfrak{M}, w \Vdash i \quad sse \quad w \in V(p),\ com\ i \in \Omega$$
$$\mathfrak{M}, w \Vdash @_i\phi \quad sse \quad \mathfrak{M}, w' \Vdash \phi,\ em\ que\ w'\ é\ a\ denotação\ de\ i$$

Como podemos perceber pela definição, para uma fórmula $@_i\phi$, o operador @ (lido como '*at*' em inglês, ou seja, 'em') se encarrega de acessar o estado i, onde ϕ será interpretado.

Outros operadores de Lógica Híbrida podem ser introduzidos, ao adicionar variáveis que varrem sobre os estados. Porém não é interessante para o nosso problema, já que provoca um aumento complexidade algumas lógicas não conservam a propriedade de decidibilidade obtida na forma mais simples [Blackburn 2000, Areces, Blackburn e Marx 1999].

Lógicas Temporais também podem ser estendidas com os operadores de Lógica Híbrida. Mostraremos a definição da Lógica HCTL(@) definida e mostrada decidível em [Benevides e Schechter 2008].

Definição 4.2.13 (Linguagem de HCTL(@))**.** *Sendo* Φ *um conjunto contável de símbolos proposicionais e* Ω *um conjunto contável de nominais. A linguagem das fórmulas de HCTL(@) é definida pela regra abaixo*

$$\phi ::= i \mid p \mid \bot \mid \neg\phi \mid (\phi \vee \phi) \mid \chi\phi \mid \mu(\phi, \phi) \mid @_i\phi$$

em que $i \in \Omega$, $p \in \Phi$, $\chi \in$ *{EX, AX}*, $\mu \in$ *{EU, AU, ER, AR}*

A semântica dos operadores dessa linguagem não difere dos apresentados anteriormente para outras lógicas.

4.3 *Model Checking*

Para aplicarmos as lógicas estudadas nesse capítulo ao problema que estamos atacando, devemos saber, como ficará claro em 5.5, a complexidade para o problema de Checagem de Modelos (do inglês *Model Checking*) nessas lógicas. Para resultados de complexidade para o problema de satisfatibilidade das lógicas apresentadas, consulte [Benevides e Schechter 2008, Kara et al. 2009, Weber 2007]. Definiremos, agora, formalmente o problema de *Model Checking* para uma linguagem temporal.

Definição 4.3.1 (Problema de decisão *Model Checking* para uma linguagem temporal $\mathcal{L}$)**.** *Seja* $\mathfrak{M} = (\mathfrak{F}, V)$ *um modelo em que* $\mathfrak{F} = (W, R)$ *é um* frame *finito e* ϕ *uma fórmula de uma linguagem temporal* $\mathcal{L}$*, em todo* $w \in W$*, temos* $\mathfrak{M}, w \vdash \phi$*?*

Vejamos, então, a complexidade desse problema nas lógicas apresentadas anteriormente.

Teorema 4.3.1 ([Sistla e Clarke 1985])**.** *O problema de* model checking *para LTL é PSPACE-completo.*

Teorema 4.3.2 ([Clarke, Emerson e Sistla 1986])**.** *O problema de* model checking *para CTL é PTIME-completo.*

Teorema 4.3.3 ([Clarke, Emerson e Sistla 1986])**.** *O problema de* model checking *para CTL* é PSPACE-completo.*

Teorema 4.3.4 ([Kupferman e Pnueli 1995])**.** *O problema de* model checking *para PCTL é PSPACE-difícil.*

Teorema 4.3.5 ([Franceschet e Rijke 2006])**.** *O problema de* model checking *para HCTL(@) é PSPACE-difícil.*

Essas complexidades dizem respeito ao problema de *model checking* para um *frame* finito arbitrário. Quando conhecemos características do frame em questão, essa complexidade pode ser reduzida. Para fazer essa análise definiremos uma restrição de 4.3.1 para classes de frames.

Definição 4.3.2 (Problema de decisão *Model Checking* para uma linguagem temporal $\mathcal{L}$ restrito a uma classe de *frames* $\boldsymbol{F}$)**.** *Seja* $\mathfrak{M} = (\mathfrak{F}, V)$ *um modelo em que* $\mathfrak{F} = (W, R)$ *é um* frame *de uma classe de* frames *finitos* F *e* ϕ *uma fórmula de uma linguagem temporal* $\mathcal{L}$*, em todo* $w \in W$*, temos* $M, w \vdash \phi$*?*

Para as lógicas LTL e CTL, podemos resolver esse problema em tempo polinomial em *frames* em formato de árvore. Vejamos os seguintes teoremas

Teorema 4.3.6 ([Kuhtz e Finkbeiner 2011])**.** *O problema de* Model Checking *para LTL restrito a modelos com* frames *finitos em formato de árvores está em* $AC^1(logDCFL)$*.*

Teorema 4.3.7 ([Kuhtz 2010])**.** *O problema de* Model Checking *para CTL restrito a modelos com* frames *finitos em formato de árvores está em* $AC^2(logDCFL)$*.*

5 VERSIONAMENTO DE ONTOLOGIAS COM LÓGICAS TEMPORAIS

Nesse capítulo exploraremos como algumas das lógicas vistas no capítulo 4 podem ser aplicadas ao Versionamento de Ontologias. Em 5.1, proporemos a estrutura do processo de versionamento, extentendo trabalhos anteriores. Já na seção 5.2, mostraremos aspectos da semântica das lógicas que proporemos. Posteriormente, em 5.3, mostraremos a nossa definição que adapta as lógicas modais a tratar de versionamento. Na seção 5.4, discutiremos vários exemplos do uso do nosso formalismo. Por último, na seção 5.5 mostraremos os resultados de complexidade obtidos para o uso do formalismo proposto com diversas lógicas.

5.1 Representação de um Versionamento

De uma forma geral, ao usar lógica para raciocinar sobre um domínio, devemos ter uma representação desse domínio. Para raciocinar sobre Versionamento de Ontologias, não será diferente. Não estamos interessados no conteúdo de cada versão, mas qual versão foi modificada para gerá-la. Nossa abordagem toma como base uma descrição de como cada versão foi gerada, ou seja, qual versão foi modificada de forma a gerar uma dada versão. Mostraremos essa noção como espaço de versões definido em [Huang e Stuckenschmidt 2005] e mostrado a seguir.

Definição 5.1.1 (Espaço de Versões)**.** *Um espaço de versões S sobre um conjunto finito de ontologias W_O é um conjunto de pares de versões, nomeadamente, $S \subseteq W_O \times W_O$.*

No nosso trabalho, a única suposição que faremos sobre a estrutura interna dos elementos de W_O é que tenham o mesmo vocabulário. Exemplificando com lógicas de descrição, tenham os mesmos nomes de papéis e conceitos. Fazemos essa suposição para evitar que um processamento de consulta falhe em uma ontologia devido a um conceito desconhecido. Logo, no nível das lógicas que estamos propondo, um elemento de W_O denota uma ontologia, mas será visto como uma estrutura indivisível onde não importa a linguagem usada para descrevê-la, nem que axiomas são usados.

Os elementos de W_O serão fundamentais para a definição de *frames* na semântica dessa lógica, sendo os próprios estados. Para manter uma distinção simbólica entre a ontologia e o estado que representa uma ontologia, usaremos o possivelmente indexado para denotar um estado e O possivelmente indexado para uma ontologia. Assim, o_i será o estado que representa a ontologia O_i. Cada par ordenado (o_i, o_j) de S representa que a ontologia O_i é ancestral imediata de O_j, ou seja, O_i foi modificada de forma a obter O_j.

5.1.1 Versionamento Linear

A abordagem adotada em [Huang e Stuckenschmidt 2005] restringe-se aos chamados Espaços de Versão Lineares. Vejamos sua definição.

Definição 5.1.2 (Espaço de Versão Linear)**.** *Um espaço de versão S sobre um conjunto de*

ontologias W_O é dito ser linear se e somente se podemos arranjar os elementos de S da seguinte forma

$$S = \{(o_1, o_2), (o_2, o_3), \ldots, (o_{n-1}, o_n)\}$$

onde $o_i \neq o_j$ para $1 \leq i < j \leq n$ e não há nenhum elemento de $o_i \in W_O$ tal que o_i ocorre em mais de dois pares de S.

Esse tipo de espaço de versões é muito importante, pois caracteriza grande parte das estruturas de versionamento de ontologias. Nesse tipo, a ontologia mais nova é fruto de modificações de uma mais antiga e sempre há somente uma ontologia mais recente. Como a própria definição nos leva imaginar, teremos uma cadeia.

Apesar de ser bastante comum, nem todas as estruturas de versionamento seguem esse padrão. Para abranger tipos diferentes de versionamento, definiremos a seguir Versionamentos Ramificado.

5.1.2 Versionamento Ramificado

Apesar do Versionamento Linear ser uma forma bastante natural para representar o histórico de versões de uma ontologia, ele não permite especificar um desenvolvimento descentralizado de uma ontologia. Vejamos como isso é nítido no seguinte exemplo.

Exemplo: Em uma aplicação, usa-se uma versão v de uma ontologia. Porém, para que a ontologia seja melhor aproveitada na aplicação, uma modificação seria necessária. Não querendo esperar por uma nova versão da ontologia, os desenvolvedores da aplicação modificam a versão v de acordo com suas necessidades, obtendo uma versão v'. Posteriormente, a ontologia é atualizada, criando uma versão v'' diferente de v'. Representaremos esse cenário resultante com a figura 5.1, em que cada ponto indica uma versão e cada aresta indica que a versão apontada foi criada a partir da que está na origem da aresta.

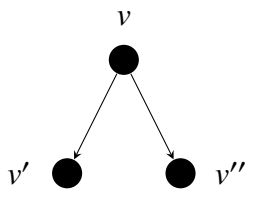

Figura 5.1: Exemplo de Versionamento Ramificado

5.1.2.1 Espaços Ramificados Considerados

Apesar de ramificações serem normais quando lidamos com o tipo de situação citada acima, não relaxaremos demais as restrições nos espaços de versionamento.

A primeira questão que iremos tratar é o quão grande pode ser um espaço de versionamento. Como discutiremos na seção 5.5, isso será determinante para sabermos a ordem de complexidade das operações que faremos. A única restrição que imporemos é que o conjunto W_O deve ser finito. Não trataremos de conjuntos infinitos de versões, pois eles não tem aplicação prática.

Finalmente, trataremos da estrutura do espaço de versionamento. Primeiro, discutiremos a possibilidade dos espaços de versão serem cíclicos como o da figura 5.2. Há necessidade desse tipo de estrutura? Esse tipo de configuração é possível em um versionamento. Porém, não é usual, já que uma nova versão indica uma necessidade de mudança que, nesse caso, teve seu resultado suprimido ao retornar a uma versão anterior. Além de não usual, esse caso seria responsável por um aumento na complexidade do sistema resultante. Por esses motivos, não iremos tratar de Espaços de Versionamento com essa forma, já que é possível desfazer os ciclos copiando o estado para o qual existe uma aresta de retorno como foi feito do lado direito da figura 5.2. Essa estratégia tem desvantagens, pois aumenta o tamanho do conjunto de estados W_O e há uma perda de informação sobre a estrutura. Porém, esse aumento não chega a um acréscimo de $|W_O|$ elementos a W_O, ou seja, o domínio não chegaria nem a dobrar de tamanho. Como veremos em 5.5, essa restrição pode evitar um aumento exponencial na complexidade do sistema, ao passo que pode piorar a complexidade do algoritmo de *model checking* que construiremos.

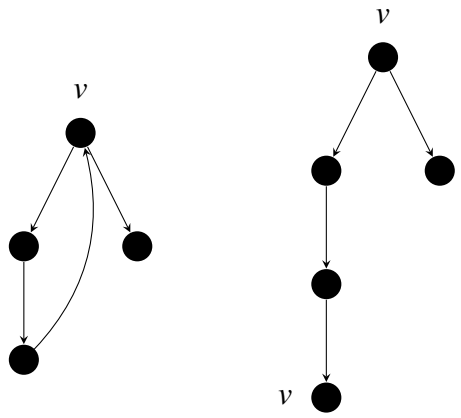

Figura 5.2: Exemplo de Espaço de Versionamento Cíclico e sua correção.

Outra restrição que faremos é quanto à conectividade do Espaço de Versionamento. Em 3.4 comentamos que Versionamento trata de como administrar versões de uma mesma ontologia e que as versões se sucedem com o tempo. Logo, temos a garantia de que não existem duas versões v' e v'' tais que não há uma versão v pertencente ao versionamento tal que v não alcança v' nem v'', ou seja, um ancestral de v' e v'' ou é igual a uma das duas arestas.

Analisando o Espaço de Versionamento como um grafo direcionado, essas restrições são suficientes para reduzirmos nosso escopo aos grafos finitos fracamente conexos e acíclicos, ou seja, grafos sem ciclos e que se ignoradas as direções das arestas, existe um caminho entre qualquer par de vértices. Isso abrange a noção de árvore foi formalizada na definição 4.1.6, bem como estruturas como a que é mostrada na figura 5.3.

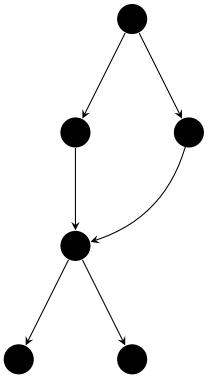

Figura 5.3: Exemplo de Espaço de Versionamento Ramificado

5.2 Frames e Modelos de Versionamento

Nessa seção, definiremos algumas noções que serão comuns às lógicas utilizadas, tomando por base os Espaços de Versionamento discutidos anteriormente.

Assim como em lógicas modais proposicionais, os modelos de lógicas modais de versionamento que estamos propondo são formados a partir de um *frame* e uma valoração. O *frame*, por sua vez, é formado por um conjunto de estados, que representarão as versões da ontologia, e uma relação de acessibilidade que indicará como o versionamento ocorreu.

Definição 5.2.1 (Frame para uma Lógica Modal de Versionamento)**.** *Um* frame $\mathfrak{F}$ *para uma linguagem modal de versionamento com espaço de versão* S *sobre um conjunto de ontologias* W_O *é definido pela tupla* $\mathfrak{F} = (W_O, S)$*.*

Definição 5.2.2 (Valoração para uma Lógica Modal de Versionamento)**.** *Uma valoração* V *para uma linguagem modal de versionamento, com linguagem de consulta* $\mathcal{L}$ *é um mapeamento* $V : W_O \to \mathcal{P}(\mathcal{L})$*, em que* $V(o)$ *representa as consultas escritas em* $\mathcal{L}$ *que são satisfeitas em* o*.*

Vale ressaltar que a valoração de um modelo não será toda definida *a priori* para qualquer consulta de uma linguagem de consulta $\mathcal{L}$, mas processada a medida que for necessário. Isso se tornará mais claro quando mostrarmos exemplos do uso dessas lógicas em 5.4

Definição 5.2.3 (Modelo para uma Lógica Modal de Versionamento)**.** *Um modelo para uma linguagem modal de versionamento é uma tupla* $\mathfrak{M} = (\mathfrak{F}, V)$*, em que* $\mathfrak{F}$ *é um* frame *para a linguagem modal de versionamento e* V *é uma valoração.*

5.3 Generalização das Lógicas Temporais para lidar com Versionamento

A capacidade de consultar uma ontologia, é fundamental para para seu uso. Tipos usuais de consulta determinam se uma ontologia satisfaz uma certa propriedade, ou quais elementos da ontologia satisfazem uma propriedade. Observe que queremos linguagens capazes de checar propriedades a respeito da estrutura do espaço de versões, bem como propriedades de uma versão de ontologia. Para que possamos fazer consultas sobre versões específicas da ontologia, nossa linguagem deve conter um fragmento de uma linguagem de consulta. Para ficar

de acordo com a semântica usual das lógicas modais, vamos escolher as consultas booleanas que podemos representar na linguagem de consulta. Poderíamos escolher um fragmento maior que o das consultas booleanas, mas, assim, não poderíamos usar lógicas modais proposicionais como base de nossa linguagem.

A definição a seguir mostrará como a intuição anterior será modelada.

Definição 5.3.1 (Linguagem Modal para modelar Versionamento de Ontologias)**.** *Seja L_M uma linguagem modal e L_C uma linguagem de consulta sobre ontologias, denotaremos por $L_M(L_C)$ a linguagem modal que, para cada consulta booleana de L_C , é designado um único símbolo proposicional de L_M. Para $L_M(L_C)$, chamaremos de linguagem L_C linguagem de consulta e L_M de linguagem modal.*

5.4 Estudos de Caso

Nessa seção mostraremos, por meio de exemplos, como as lógicas modais generalizadas pela definição 5.3.1 podem ser usadas para expressar propriedades de um espaço de versões.

5.4.1 Lógica Modal Básica

Primeiramente, vamos mostrar um exemplo com a Lógica Modal Básica que abreviaremos como LMB, a lógica mais simples vista e a Lógica de Descrição $\mathcal{ALC}$ como lógica de processamento de consultas. A Lógica Modal de Versionamento seria chamada, então, de LMB($\mathcal{ALC}$).

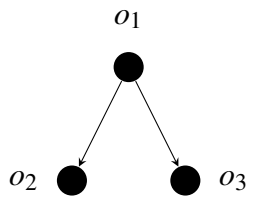

Figura 5.4: Exemplo de Espaço de Versionamento

Sejam $O_1 = \{A \sqsubseteq B, B \sqsubseteq C\}$, $O_2 = \{A \sqsubseteq B, B \sqsubseteq C, D \sqsubseteq B\}$, $O_3 = \{A \sqsubseteq C, B \sqsubseteq C\}$ as ontologias descritas em $\mathcal{ALC}$ denotadas por o_1, o_2 e o_3 na figura 5.4. A seguir, vamos mostrar a satisfação de algumas fórmulas no espaço de versionamento da figura 5.4 que chamaremos de $\mathfrak{M}_{5.4}$.

$$\mathfrak{M}_{5.4}, o_1 \Vdash \Diamond(A \sqsubseteq B) \tag{5.1}$$
$$\mathfrak{M}_{5.4}, o_2 \Vdash (A \sqsubseteq B) \tag{5.2}$$
$$\mathfrak{M}_{5.4}, o_1 \nVdash \Box(A \sqsubseteq B) \tag{5.3}$$
$$\mathfrak{M}_{5.4}, o_1 \Vdash \Box((A \sqsubseteq B) \rightarrow (B \sqsubseteq C)) \tag{5.4}$$
$$\mathfrak{M}_{5.4}, o_1 \Vdash \Diamond(D \sqsubseteq A) \tag{5.5}$$

Nas fórmulas acima, é possível verificar como ocorre o uso de consultas de $\mathcal{ALC}$ como símbolos proposicionais de uma linguagem modal. Note que é possível expressar propriedades simples a respeito das versões que sucedem uma determinada ontologia. Assim é possível escolher uma determinada versão, baseando-se nas suas propriedades e nas de versões que a sucedem.

Ao utilizar a LMB conseguimos expressar algumas propriedades do espaço de versionamento. Apesar disso sua expressividade é muito limitada para lidar com propriedades mais complexas, como expressar as propriedades de um determinado caminho. A seguir, mostraremos como a lógica LTL pode ser usada para isso.

5.4.2 LTL

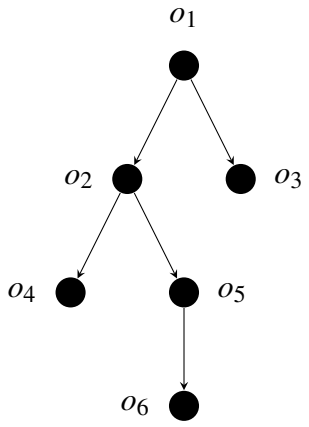

Figura 5.5: Exemplo de Espaço de Versionamento

Para o exemplo da figura 5.5, usaremos O_1,O_2, O_3 conforme $\mathfrak{M}_{5.4}$ e $O_4 = \{A \sqsubseteq C, B \sqsubseteq C, A \sqcap B \sqsubseteq \bot\}$, $O_5 = \{A \sqsubseteq C, B \sqsubseteq C, D \sqsubseteq A, D \sqcap B \sqsubseteq \bot\}$, $O_6 = \{A \sqsubseteq C, B \sqsubseteq C, D \sqsubseteq A, A \sqcap B \sqsubseteq \bot\}$. Mostraremos a seguir exemplos de fórmulas de LTL($\mathcal{ALC}$).

$$\mathfrak{M}_{5.5}, o_1 \nVdash AF(A \sqsubseteq B) \tag{5.6}$$
$$\mathfrak{M}_{5.5}, o_2 \Vdash (A \sqsubseteq B) \tag{5.7}$$
$$\mathfrak{M}_{5.5}, o_1 \Vdash AF(A \sqsubseteq C) \tag{5.8}$$
$$\mathfrak{M}_{5.5}, o_1 \Vdash AFG(D \sqsubseteq A) \tag{5.9}$$
$$\mathfrak{M}_{5.5}, o_1 \nVdash A(F(D \sqsubseteq B) \rightarrow F(D \sqsubseteq A)) \tag{5.10}$$

Apesar de LTL oferecer uma sintaxe própria para falar de caminhos, por não haver o quantificador E, essa lógica não é expressiva o suficiente para designar a propriedade da existência de um estado com uma certa característica. Algumas fórmulas desse tipo podem ser obtidas na linguagem modal básica(LMB) como utilizado em 5.1 do primeiro exemplo. Porém, com o quantificador de caminho A de LTL conseguimos expressar facilmente propriedades de todos os caminhos do modelo como em 5.8 do exemplo anterior. Através da aplicação de sucessivas modalidades, pelos modelos examinados serem finitos é possível expressar algumas fórmulas desse tipo em LMB, mas o tamanho da fórmula tende a crescer muito, aumentando a complexidade do *Model Checking*. Já a fórmula de 5.9 do exemplo anterior não é expressada em LMB. Nesse caso tomamos vantagem do fato de todas as ontologias terem o mesmo vocabulário, como chamamos atenção anteriormente.

Com LTL ganhamos formas bastante simples para expressar propriedades de caminho. Porém só podemos atestar propriedades de todos os caminhos. A lógica CTL pode ser usada para expressar propriedades da existência de caminhos. Vejamos como usá-la em conjunto com uma linguagem de consulta.

5.4.3 CTL

Como já apresentado no capítulo 4, CTL e LTL tem expressivades não comparáveis, ou seja, existem fórmulas de CTL que não são expressíveis em LTL e vice-versa. Usaremos o mesmo exemplo da figura 5.5 para demonstrar algumas fórmulas de CTL($\mathcal{ALC}$).

$$\mathfrak{M}_{5.5}, o_1 \Vdash EG(A \sqcap B \sqsubseteq \bot) \tag{5.11}$$
$$\mathfrak{M}_{5.5}, o_1 \Vdash AF(A \sqsubseteq C) \tag{5.12}$$
$$\mathfrak{M}_{5.5}, o_1 \Vdash AF(D \sqsubseteq B) \rightarrow AF(D \sqsubseteq A) \tag{5.13}$$

Vale chamar atenção para a fórmula 5.13. Apesar da semelhança com a fórmula 5.10, elas não expressam a mesma propriedade. O modelo $\mathfrak{M}_{5.5}$ serve de exemplo, já que o valor verdade delas é diferente quando aplicado ao mesmo estado. Isso mostra que apesar de ganharmos poder expressivo ao adicionar o quantificador E, perdemos poder expressivo quando não fixamos que um operador modal temporal deve ocorrer prescedido de um quantificador de caminho.

5.4.4 CTL*

Também como discutimos no capítulo 4, CTL* apresenta expressividade extritamente maior que as Lógicas Anteriores. No exemplo a seguir, construimos uma fórmula de CTL*($\mathcal{ALC}$) que não pode ser expressa em LTL($\mathcal{ALC}$) nem em CTL($\mathcal{ALC}$).

$$\mathfrak{M}_{5.5}, o_1 \Vdash (EG(A \sqcap B \sqsubseteq \bot)) \wedge \neg A(F(D \sqsubseteq B) \rightarrow F(D \sqsubseteq A)) \tag{5.14}$$

5.5 Complexidade

Como é possível notar em 5.4, resolver o problema de *model checking* para uma lógica $L_M(L_C)$ de forma eficiente é essencial para o uso do modelo de versionamento proposto. Pela definição 5.3.1, podemos notar que podemos obter um algoritmo de *model checking* para essa lógica através de uma modificação do algoritmo de *model checking* de L_M. Ao ser requisitado o valor verdade de um símbolo proposicional, seria processada a consulta booleana equivalente em L_C. Logo, a complexidade do *model checking* de $L_M(L_C)$ depende da complexidade do algoritmo de model checking de L_M e a complexidade de processamento de consultas de L_C.

A seguir, apresentaremos um teorema que ajuda a determinar a complexidade do *model checking* da lógica modal generalizada resultante.

Teorema 5.5.1. *Seja L_M uma linguagem modal cujo problema de* model checking *está em uma classe $DTIME(f)$ e L_C uma linguagem de consulta de ontologias tal que o processamento de uma consulta dessa linguagem está em $DTIME(g)$. O* model checking *da lógica $L_M(L_C)$ com entrada φ e modelo $\mathfrak{M} = (\mathfrak{F}, V)$ com* frame *$\mathfrak{F} = (W_O, S)$, em que a maior consulta que ocorre em φ tem tamanho $|Q|$ e cada ontologia correspondente a um elemento de W_O tem tamanho máximo de $|O|$, está na classe de complexidade $DTIME(f(|S|, |\varphi|) + |S| * |\varphi| * g(|O|, |Q|))$.*

Demonstração. Construiremos um algoritmo de *model checking* de $L_M(L_C)$ modificando o algoritmo de *model checking* de L_M. No nosso algoritmo, devido à sintaxe de $L_M(L_C)$, cada checagem de valor verdade de símbolo proposicional do algoritmo de L_M será substituído por um processamento de consulta da linguagem L_C. Assim, teremos no máximo $|\varphi| * |S|$ chamadas de processamento de consultas. Isso corresponde ao pior caso do número de chamadas de consultas, que ocorre quando φ é composta somente de consultas a serem checadas em todos os estados de S.

O processamento de consultas é somente uma parte da complexidade do nosso algoritmo. Devemos considerar, também, o custo de executar o restante do algoritmo que resolve o *model checking* para L_M que é aproveitado integralmente para o algoritmo de *model checking* de $L_M(L_C)$.

Considerando essas duas parcelas, obtemos que resolver o *model checking* de $L_M(L_C)$ está em $DTIME(f(|S|, |\varphi|) + |S| * |\varphi| * g(|O|, |Q|))$. □

Analisando o teorema 5.5.1, podemos notar que para uma linguagem L_M com complexidade de *model checking* suficientemente baixa, o formalismo de raciocínio sobre versões estará na mesma classe de complexidade do processamento de consultas de L_C. Assim, ao usar o formalismo proposto, não há uma sobrecarga de processamento muito grande em comparação com o processamento de consultas de L_C. Na tabela 5.4 mostraremos algumas aplicações do teorema 5.5.1 a algumas linguagens modais que apresentamos no capítulo 4 e de processamento de consultas. Isso permitirá uma melhor compreensão da ideia exposta acima. Antes, em 5.1 mostraremos as classes de complexidade do problema de processamento de consultas nas linguagens que exporemos na tabela 5.4.

Linguagem	Classe de Complexidade
$\mathcal{ALC}$	PSPACE-completo [Schmidt-Schauß e Smolka 1991]
SPARQL+RDF	PSPACE-completo [Pérez, Arenas e Gutierrez 2009]
OWL DL	NEXPTIME-completo [Horrocks e Patel-Schneider 2003]
OWL 2 QL	NP-completo [Motik et al. 2009]

Tabela 5.1: Complexidades de Processamento de Consultas

Linguagem	Árvore	Grafo Fracamente Conexos e Acíclicos
LTL	$AC^1(logDCFL)$	$AC^1(logDCFL)$
CTL	$AC^2(logDCFL)$	PTIME
CTL*	PSPACE	PSPACE
HCTL	PSPACE-difícil	PSPACE-difícil
PCTL	PSPACE-difícil	PSPACE-difícil

Tabela 5.2: Classes de Complexidades de *Model Checking* de acordo com o tipo de *frame*

L_M \ L_C	$\mathcal{ALC}$	SPARQL+RDF	OWL DL	OWL 2 QL
LTL	PSPACE-completo	PSPACE-completo	NEXPTIME-completo	NP-completo
CTL	PSPACE-completo	PSPACE-completo	NEXPTIME-completo	NP-completo
CTL*	PSPACE-completo	PSPACE-completo	NEXPTIME-completo	PSPACE
HCTL	PSPACE-completo	PSPACE-completo	NEXPTIME-completo	PSPACE-difícil
PCTL	PSPACE-completo	PSPACE-completo	NEXPTIME-completo	PSPACE-difícil

Tabela 5.3: Complexidades das Lógicas Modais de Versionamento para árvores

L_M \ L_C	$\mathcal{ALC}$	SPARQL+RDF	OWL DL	OWL 2 QL
LTL	PSPACE-completo	PSPACE-completo	NEXPTIME-completo	NP-completo
CTL	PSPACE-completo	PSPACE-completo	NEXPTIME-completo	NP-completo
CTL*	PSPACE-completo	PSPACE-completo	NEXPTIME-completo	PSPACE
HCTL	PSPACE-completo	PSPACE-completo	NEXPTIME-completo	PSPACE-difícil
PCTL	PSPACE-completo	PSPACE-completo	NEXPTIME-completo	PSPACE-difícil

Tabela 5.4: Complexidades das Lógicas Modais de Versionamento para grafos fracamente conexos acíclicos

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

O modelo de Versionamento de Ontologias utilizado atualmente limita que somente uma versão é obtida a partir de uma modificação de uma versão anterior [Huang e Stuckenschmidt 2005, Flouris et al. 2008]. Nesse trabalho, damos uma nova perspectiva sobre como versionar uma Ontologia. Propomos um modelo que aceite que mais de uma versão seja gerada, abrangendo um desenvolvimento não regulado. Para isso, consideramos espaços de versões não lineares, assumimos espaços acíclicos e fracamente conexos, mas também analisamos o caso de árvore. Até onde sabemos, nosso trabalho é o único que fornece um formalismo lógico para tratar do desenvolvimento não regulado de ontologias. Propomos uma generalização de lógicas modais temporais para atestar propriedades de um espaço de versionamento. Disponibilizamos estudos de expressividade e complexidade de diversas linguagens modais que serão utilizadas para realizar consultas sobre o espaço de versões. Obtemos, também, um resultado que determina um limite superior da medida de complexidade do arcabouço lógico gerado a partir de uma linguagem modal e uma linguagem de consulta.

Apesar desses resultados de complexidade, um trabalho futuro poderia ser a investigação de formas de diminuir a complexidade do sistema ao ser implementado. Diversas abordagens podem ser tentadas. Poderíamos tentar diminuir a complexidade total do sistema restringindo o espaço de versões eliminando versões indesejadas. Isso poderia ser feito através de condições que uma dada versão deveria satisfazer para ter sua entrada no sistema. Outra maneira é dar essa escolha ao usuário, permitindo que descarte ramos que não o interessam. Detectar versões de um espaço de versões equivalentes segundo alguma comparação também pode ser um caminho. Outra alternativa para reduzir a complexidade seria efetuar menos consultas. Uma forma seria pré-computar propriedades das versões, de forma a diminuir a carga computacional para próximas chamadas do algoritmo de *model checking*. Informações adicionais à respeito das mudanças de uma versão para outra também pode ajudar na redução de complexidade. Por exemplo, se soubermos *a priori* as modificações que resultaram na criação de uma dada versão, podemos aproveitar o que obtemos para uma versão anterior e responder consultas sem a necessidade de processá-las.

Os resultados obtidos permitem que os desenvolvedores de aplicações tenham autonomia na escolha de qual versão de uma ontologia usar, seja em um desenvolvimento regulado ou não, ao poder atestar suas propriedades em comparação com outras.

Esse formalismo é aplicável somente aos problemas de escolha e comparação de versões de uma ontologias. Um outro trabalho futuro poderia ser investigar a adaptação desse formalismo a outros problemas de versionamento ou de outras subáreas de mudança de ontologias. Uma possível adaptação seria definir uma forma mais fraca de bissimulação para comparar espaços de versionamentos e com isso tentar resolver problemas de heterogeneidades de vocabulário. A bissimulação também poderia ser usada como forma de resover o problema de identificar se duas versões são diferentes de um ponto de vista semântico, analisando-o juntamente com as versões que são desenvolvidas a partir dela.

REFERÊNCIAS BIBLIOGRÁFICAS

ANTONIOU, G.; HARMELEN, F. V. *A Semantic Web Primer*. [S.l.]: Mit Press, 2004. (Cooperative Information Systems). ISBN 9780262012102.

ARECES, C.; BLACKBURN, P.; MARX, M. A Road-Map on Complexity for Hybrid Logics. In: FLUM, J.; RODRIGUEZ-ARTALEJO, M. (Ed.). *Computer Science Logic*. [S.l.]: Springer Berlin / Heidelberg, 1999, (Lecture Notes in Computer Science, v. 1683). p. 826–826. ISBN 978-3-540-66536-6.

ARORA, S.; BARAK, B. *Computational Complexity - A Modern Approach*. [S.l.]: Cambridge University Press, 2009. I–XXIV, 1–579 p. ISBN 978-0-521-42426-4.

BAADER, F. et al. (Ed.). *The description logic handbook: theory, implementation, and applications*. New York, NY, USA: Cambridge University Press, 2003. ISBN 0-521-78176-0.

BAUER, A.; LEUCKER, M.; SCHALLHART, C. The good, the bad, and the ugly, but how ugly is ugly? In: *Proceedings of the 7th international conference on Runtime verification*. Berlin, Heidelberg: Springer-Verlag, 2007. (RV'07), p. 126–138. ISBN 3-540-77394-0, 978-3-540-77394-8. Disponível em: <http://dl.acm.org/citation.cfm?id=1785141.1785155>.

BAUER, A.; LEUCKER, M.; SCHALLHART, C. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 2009. Disponível em: <http://logcom.oxfordjournals.org/content/early/2010/02/04/logcom.exn075.abstract>.

BENEVIDES, M.; SCHECHTER, L. M. Decidability of a Syntactic Fragment of the Hybrid Computation Tree Logic with the Downarrow Operator. In: *EBL*. [S.l.: s.n.], 2008.

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. The Semantic Web (Berners-Lee et. al 2001). maio 2001.

BLACKBURN, P. Representation, Reasoning, and Relational Structures: a Hybrid Logic Manifesto. *Logic Journal of IGPL*, v. 8, p. 2000, 2000.

BLACKBURN, P.; RIJKE, M.; VENEMA, Y. *Modal logic*. [S.l.]: Cambridge University Press, 2002. (Cambridge tracts in theoretical computer science). ISBN 9780521527149.

CLARKE, E.; DRAGHICESCU, I. Expressibility results for linear-time and branching-time logics. In: BAKKER, J. de; ROEVER, W. de; ROZENBERG, G. (Ed.). *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*. Springer Berlin / Heidelberg, 1989, (Lecture Notes in Computer Science, v. 354). p. 428–437. ISBN 978-3-540-51080-2. 10.1007/BFb0013029. Disponível em: <http://dx.doi.org/10.1007/BFb0013029>.

CLARKE, E.; GRUMBERG, O.; PELED, D. *Model checking*. [S.l.]: MIT Press, 1999. ISBN 9780262032704.

CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, v. 8, p. 244–263, 1986.

EMERSON, E. A.; CLARKE, E. M. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Sci. Comput. Program.*, v. 2, n. 3, p. 241–266, 1982. Disponível em: <http://dx.doi.org/10.1016/0167-6423(83)90017-5>.

FLOURIS, G. *On Belief Change and Ontology Evolution*. Tese (Doutorado) — Computer Science Department, University of Greece, Greece, 2006.

FLOURIS, G. et al. Ontology change: classification and survey. *Knowledge Eng. Review*, v. 23, n. 2, p. 117–152, 2008.

FLOURIS, G.; PLEXOUSAKIS, D. *Handling Ontology Change: Survey and Proposal for a Future Research Direction*. [S.l.], 2005.

FRANCESCHET, M.; RIJKE, M. de. Model checking for hybrid logics (with an application to semistructured data). *Journal of Applied Logic*, v. 4, n. 3, p. 279–304, 2006.

GABBAY, D. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In: *Proceedings 1st Conference on Temporal Logic in Specification (TLS'1987)*. [S.l.: s.n.], 1987. (Lecture Notes in Computer Science), p. 409–448.

GABBAY, D. et al. On the temporal analysis of fairness. In: *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1980. (POPL '80), p. 163–173. ISBN 0-89791-011-7.

GORANKO, V.; MONTANARI, A.; SCIAVICCO, G. A Road Map of Interval Temporal Logics and Duration Calculi. *Journal of Applied Non-Classical Logics*, v. 14, n. 1-2, p. 9–54, 2004.

GRUBER, T. R. A translation approach to portable ontology specifications. *Knowl. Acquis.*, Academic Press Ltd., London, UK, UK, v. 5, n. 2, p. 199–220, jun. 1993. ISSN 1042-8143. Disponível em: <http://dx.doi.org/10.1006/knac.1993.1008>.

HEFLIN, J.; HENDLER, J.; LUKE, S. Coping with Changing Ontologies in a Distributed Environment. In: *In Proceedings of AAAI-99 Workshop on Ontology Management*. [S.l.]: Press, 1999. p. 74–79.

HORROCKS, I.; PATEL-SCHNEIDER, P. F. Reducing OWL Entailment to Description Logic Satisfiability. In: FENSEL, D.; SYCARA, K.; MYLOPOULOS, J. (Ed.). *Proc. of the 2nd International Semantic Web Conference (ISWC 2003)*. [S.l.]: Springer, 2003. (Lecture Notes in Computer Science, v. 2870), p. 17–29. ISBN 3-540-20362-1.

HUANG, Z.; STUCKENSCHMIDT, H. Reasoning with Multi-Version Ontologies: A Temporal Logic Approach. In: *In Proceeding of the 4th International Semantic Web Conference (ISWC)*. [S.l.: s.n.], 2005. p. 398–412.

KARA, A. et al. On the Hybrid Extension of CTL and CTL+. *CoRR*, abs/0906.2541, 2009.

KLEIN, M.; FENSEL, D. Ontology versioning on the Semantic Web. In: *Stanford University*. [S.l.: s.n.], 2001. p. 75–91.

KLEIN, M. et al. Ontology versioning and change detection on the Web. In: *In 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*. [S.l.: s.n.], 2002. p. 197–212.

KUHTZ, L. *Model checking finite paths and trees.* Tese (Doutorado) — Saarland University, 2010.

KUHTZ, L.; FINKBEINER, B. Weak Kripke Structures and LTL. In: *CONCUR*. [S.l.: s.n.], 2011. p. 419–433.

KUPFERMAN, O.; PNUELI, A. Once and For All. In: *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, 1995. p. 25–35. ISBN 0-8186-7050-6.

LAROUSSINIE, F.; MARKEY, N.; SCHNOEBELEN, P. Temporal Logic with Forgettable Past. In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, 2002. (LICS '02), p. 383–392. ISBN 0-7695-1483-9.

LAROUSSINIE, F.; SCHNOEBELEN, P. A hierarchy of temporal logics with past. *Theor. Comput. Sci.*, Elsevier Science Publishers Ltd., Essex, UK, v. 148, n. 2, p. 303–324, September 1995. ISSN 0304-3975.

LIU, H. et al. Updating Description Logic ABoxes. In: DOHERTY, P.; MYLOPOULOS, J.; WELTY, C. (Ed.). *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*. [S.l.]: AAAI Press, 2006. p. 46–56.

MANNA, Z.; PNUELI, A. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995. (Temporal verification of reactive systems / Zohar Manna; Amir Pnueli). ISBN 9780387944593. Disponível em: <http://books.google.at/books?id=McYxCfcpGkoC>.

MCGUINNESS, D. L. et al. Daml+oil: An ontology language for the semantic web. *IEEE Intelligent Systems*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 17, n. 5, p. 72–80, set. 2002. ISSN 1541-1672. Disponível em: <http://dx.doi.org/10.1109/MIS.2002.1039835>.

MORGENSTERN, A.; GESELL, M.; SCHNEIDER, K. An Asymptotically Correct Finite Path Semantics for LTL. In: *LPAR*. [S.l.: s.n.], 2012. p. 304–319.

MOTIK, B. et al. *OWL 2 Web Ontology Language – Profiles*. [S.l.], 2009.

NOY, N. F.; MUSEN, M. A. PROMPTDIFF: A Fixed-Point Algorithm for Comparing Ontology Versions. In: *AAAI/IAAI'02*. [S.l.: s.n.], 2002. p. 744–750.

PAPADIMITRIOU, C. H. *Computational complexity*. [S.l.]: Addison-Wesley, 1994. I–XV, 1–523 p. ISBN 978-0-201-53082-7.

PÉREZ, J.; ARENAS, M.; GUTIERREZ, C. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, ACM, New York, NY, USA, v. 34, n. 3, p. 16:1–16:45, set. 2009. ISSN 0362-5915. Disponível em: <http://doi.acm.org/10.1145/1567274.1567278>.

PNUELI, A. The Temporal Logic of Programs. In: *FOCS*. [S.l.: s.n.], 1977. p. 46–57.

PRIOR, A. *Time and modality*. [S.l.]: Clarendon Press, 1957. (John Locke lectures).

QI, G.; LIU, W.; BELL, D. A revision-based approach to handling inconsistency in description logics. *Artificial Intelligence Review*, Springer Netherlands, v. 26, n. 1, p. 115–128, 2006. ISSN 0269-2821.

RIBEIRO, M. M. et al. Belief contraction in web-ontology languages. In: *Proceedings of the International Workshop on Ontology Dynamics (IWOD-2009)*. [S.l.: s.n.], 2009.

SCHMIDT-SCHAUSS, M.; SMOLKA, G. Attributive concept descriptions with complements. *Artificial Intelligence*, v. 48, n. 1, p. 1–26, 1991. ISSN 0004-3702.

SISTLA, A. P.; CLARKE, E. M. The complexity of propositional linear temporal logics. *J. ACM*, ACM, New York, NY, USA, v. 32, n. 3, p. 733–749, July 1985. ISSN 0004-5411.

SISTLA, A. P.; VARDI, M. Y.; WOLPER, P. The Complementation Problem for Büchi Automata with Applications to Temporal Logic. *Automata Languages and Programming*, Lecture Notes in Computer Science, Springer-Verlag, v. 49, p. 217–237, 1987.

VOLLMER, H. *Introduction to Circuit Complexity: A Uniform Approach*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. ISBN 3540643109.

WEBER, V. Hybrid Branching-Time Logics. *CoRR*, abs/0708.1723, 2007.