Universidade Federal do Amazonas
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programa de Pós-Graduação em Informática

SwTO^{I} (Software Test Ontology Integrated): uma Ontologia com Aplicação em Teste do Linux

Daniella Rodrigues Bezerra

Manaus
2008

Universidade Federal do Amazonas
Instituto de Ciências Exatas

Daniella Rodrigues Bezerra

SwTO^I (*Software Test Ontology Integrated*): uma Ontologia com Aplicação em Teste do Linux

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática do Instituto de Ciências Exatas da UFAM, como parte dos requisitos necessários para a obtenção do título de Mestre em Informática. Área de concentração: **Inteligência Artificial**.

Orientadora: Profª Virgínia Brilhante, PhD.

Manaus
2008

Daniella Rodrigues Bezerra

SwTO^{I} (Software Test Ontology): uma Ontologia com Aplicação em Teste do Linux

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática do Instituto de Ciências Exatas da UFAM, como parte dos requisitos necessários para a obtenção do título de Mestre em Informática. Área de concentração: **Inteligência Artificial**.

BANCA EXAMINADORA

Profª Virgínia Brilhante, PhD., Presidente
Universidade Federal do Amazonas

Prof. Ilias Biris, PhD., Membro
Nokia Corporation

Prof. Fred Freitas, Dr., Membro
Universidade Federal de Pernambuco

Ao meu pai e à minha mãe por plantarem a semente. À vida pelo solo fértil. À todos da minha família pelo incentivo para realização deste trabalho.

Agradecimentos

Ao plano espiritual pela força invisível;

Ao Edjar e Mônica pela orientação inicial;

À minha orientadora "brilhante"e ao Ilías pelo acompanhamento constante;

Aos colegas Afonso, João Leite e Rodrigo;

À minha mãe e verdadeira amiga Olívia pelo seu carinho, paciência e conselhos;

Ao meu tio Mário, tia Clenes e tia Cleomar pelo apoio e pela torcida organizada;

Aos meus avós pelo carinho e preocupação;

À FAPEAM pela oportunidade e concessão da bolsa de estudos;

Ao meu namorado Valter Júnior pelo carinho;

À minha amiga Eline pelo seu apoio e orações;

Aos amigos Meg, Coração, Lano, Clara e Teko pelos momentos de descontração;

Aos colegas Vitor, Manu, Gustavo, Márcio, Juan e Patrícia;

À Elienai e equipe pelo apoio durante o mestrado;

À UFAM e a todos os professores pela experiência de vida que o mestrado me proporcionou.

Que vocês sejam grandes empreendedores. Se empreenderem, não tenham medo de falhar. Se falharem, não tenham medo de chorar. Se chorarem, repensem a sua vida, mas não desistam. Dêem sempre uma nova chance a si mesmos.

Augusto Cury

O conhecimento pronto estanca o saber e a dúvida provoca a inteligência.

Lev Vigotsky

Resumo

Este trabalho reúne elementos de um estudo sobre representação do conhecimento fundamentado em ontologias tendo como domínio alvo o teste do Linux. O estudo visa demonstrar que uma vez que o conhecimento é formalizado, é possível reusá-lo, realizar inferência, processá-lo computacionalmente, como também torna-se passível de comunicação entre pessoas e *software*. Para tal, foram desenvolvidas três ontologias: a OSOnto (*Operating System Ontology*) que representa conceitos do domínio de Sistema Operacional, a SwTO (*Software Test Ontology*) que trata do domínio de teste de *software*, e a SwTO^I (SwTO *Integrated*) que representa conceitos destes dois domínios integrados. Para a implementação das ontologias foi utilizada OWL DL como linguagem de especificação, o Protégé como ambiente de edição e o Racer como principal raciocinador. Uma avaliação quantitativa e qualitativa foi realizada da ontologia SwTO^I.

Abstract

This work encompasses elements of a study of knowlwedge representation founded on ontologies that have Linux testing as target domain. The study aims at demonstrating that once knowledge is formalised, it is possible to reuse it, to perform inference, to process it throuch computers, and, what is more, it becames amenable to being communicated between people and software. Towards that, three ontlogies have been developed: OSOnto (Operating System Ontology) which represents concepts of the operating system domain, SwTO (Software Test Ontology) which deals with the software testing domain, and SwTO^I (SwTO Integrated) which represents concepts of both the above domains in an integrated way. For implementing the ontologies, OWL DL as ontology specification language, Protégé as ontology edition environment and Racer as the main reasoner, have been used. A quantitative and qualitative evaluation of the SwTO^I ontology has been performed.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A manifestação inteligente do conhecimento pressupõe aquisição, armazenamento e inferência. Grande parte do esforço em Inteligência Artificial (IA) tem se concentrado em buscar e aperfeiçoar formalismos para a representação do conhecimento [5, 32].

Ontologia é um dos formalismos existentes para a representação do conhecimento. Este paradigma representa a epistemologia aplicada. Quando uma ontologia faz uso de uma linguagem lógica, esta passa a ser reconhecida como uma ontologia formal, agregando vários benefícios à representação como a eliminação de ambigüidades, corretude, reuso e inferência sobre o domínio.

1.1 Motivação e Justificativa

Um dos grandes desafios que a Engenharia de Software enfrenta é: como construir *software* de boa qualidade com o menor custo, tempo e esforço possível? Várias soluções tecnológicas são apontados como resposta para esta pergunta. Uma delas é o Software Livre que procura contribuir com seus exemplos de projetos colaborativos fundamentados na liberdade do conhecimento e da propriedade intelectual.

Dentre os vários projetos de *software* livre, o Linux se destaca por ter sido um dos pioneiros, por reunir um número significativo de colaboradores e usuários em todo o mundo e pela sua qualidade decorrente da maturidade do projeto e do corpo técnico, formado por pessoas que participam ativamente. Entretanto, a qualidade do Linux é muito mais observada e sentida por seus usuários do que efetivamente comprovada por procedimentos científicos.

Com a inserção do Linux no domínio corporativo, a qualidade precisa ser efetivamente comprovada e isso fez com que as comunidades e empresas interessadas se unissem, dando origem ao *Linux Test Project* (LTP). Trata-se de um projeto cujo foco é garantir a qualidade do Linux por intermédio de testes sistemáticos. Este projeto comporta um repositório com mais de 3.000 casos de teste, *scripts* e ferramentas. Parte da documentação existente para o processo de teste do LTP é deficiente e não há um registro eficiente do conhecimento dos projetistas na elaboração dos testes.

Estas dificuldades identificadas dentro da comunidade de teste do Linux é comum em muitos outros projetos, não só de *software* livre e isso motivou a investigação deste domínio e despertou o interesse de oferecer uma contribuição que pelo menos reduza estas dificuldades.

A principal justificativa para realização deste trabalho é contribuir com a comunidade científica através da compilação do estudo sobre formalização do conhecimento por intermédio de ontologias formais, discutir sobre o teste no paradigna de *software* livre por intermédio do Linux como amostra de projeto, bem como apontar um formalismo capaz de melhorar a comunição entre equipes de desenvolvimento e trazer benefícos para a produção de *software* com alta qualidade.

1.2 Objetivos

O presente trabalho propõe a representação do conhecimento referente ao teste do Linux, fundamentada em ontologias. Esta representação tem como objetivo oferecer uma perspectiva aplicável à representação e análise de teste como mecanismo de aprimoramento da qualidade do Linux, de forma a sustentar a implementação de sistemas de informação voltados para o domínio de teste do Linux.

Em linhas gerais, este objetivo principal pode ser detalhado da seguinte forma:

1. *Representar o conhecimento sobre o teste do Linux, apresentando uma parte do conhecimento científico sobre o domínio por intermédio de uma ontologia;*

2. *Fornecer um vocabulário formal que represente inicialmente o consenso de um grupo de pesquisa e posteriormente, possa ser disponibilizado e amadurecido com a participação dinâmica de comunidades que tiverem interesse sobre o conhecimento do domínio;*

3. *Ilustrar o uso do conhecimento representado no domínio de teste do Linux, mostrando como ferramentas podem fazer uso deste conhecimento;*

4. *Avaliar a ontologia desenvolvida por meio de uma análise quantitativa e qualitativa.*

1.3 Linha de Pesquisa

A linha de pesquisa que conduziu este trabalho foi inicialmente investigar o domínio de teste do Linux e o LTP como repositório de teste. Posteriormente, investigar um formalismo lógico que permitisse a implementação de uma ontologia formal (ou pesada).

Quanto à generalidade, foi construir uma ontologia híbrida entre domínio e aplicação, de forma que contivesse conceitos de um domínio específico de conhecimento (teste de *software* e sistema operacional) mas que também atendesse à solução de um problema específico (critérios de elaboração de teste sistemático, definição de um vocabulário formal, registro eficiente do conhecimento dos projetistas de teste sobre suas decisões de cobertura do Linux)

1.4 Métodos, Técnicas e Recursos de *Software*

Para atingir os objetivos especificados, foram utilizados vários métodos e técnicas. Primeiramente, para análise do domínio foram feitas pesquisas bibliográficas e entrevistas com especialistas do domínio.

De posse dessas informações, deu-se início ao processo de desenvolvimento da ontologia seguindo o método de Uschold e Gruninger [37] que define passos para a construção de ontologias.

A linguagem OWL DL foi escolhida para a implementação da ontologia. O Protégé foi utilizado como ambiente de edição e como raciocinadores, o Racer e o Pellet.

A gerência de configuração do código-fonte bem como de toda a documentação gerada foi feita pelo SVN.

1.5 Organização da Dissertação

Esta dissertação está dividida em três partes. A Parte I, "Fundamentação Teórica", corresponde ao estudo realizado e concentra as referências mais relevantes para este trabalho. O Capítulo 2 trata das questões de qualidade e teste no contexto do *software* livre e apresenta dois projetos significativos para este trabalho:

o Linux e o LTP. O Capítulo 3 destaca os benefícios de uma representação formal de conhecimento e de DL (*Description Logic*) como formalismo lógico. A Parte II, trata da orientação metodológica utilizada para a condução deste trabalho. O Capítulo 4, destaca o problema, a linha de pesquisa do mesmo, bem como a caracterização do estudo e os métodos e técnicas utilizados de forma que o estudo possa ser repetido por outros pesquisadores. A Parte III apresenta os resultados obtidos pelo trabalho bem como uma análise e interpretação dos mesmos. Optou-se por apresentar separadamente a interpretação dos resultados da análise já que esta é extensa. Essa divisão evita que interpretações pessoais se misturem com a análise. Esta parte envolve três capítulos. O Capítulo 5 apresenta a análise do conhecimento representado sobre o domínio de teste de *software* integrado ao domínio de sistemas operacionais. O Capítulo 6 analisa e critica a utilização da Ontologia desenvolvida na aplicação TeSG. O Capítulo 7 valida a ontologia desenvolvida através de uma avaliação quantitativa e qualitativa, apontando características da Ontologia. A conclusão do trabalho é apresentada no Capítulo 8 seguida pelas referências utilizadas no desenvolvimento deste. Para finalizar, o Apêndice apresenta a OSOnto, uma ontologia de Sistema Operacional.

Parte I

Fundamentação Teórica

Capítulo 2

O Teste de *Software* Livre

Este capítulo procura responder as seguintes questões: Quais são os fatores que influenciam na qualidade do *software* livre? Que técnicas de qualidade estão sendo utilizadas pelos desenvolvedores no processo de desenvolvimento de *software* livre? Qual a percepção dos desenvolvedores e da comunidade de *software* livre acerca da qualidade? Que tipo de contribuição as comunidades aguardam? Como os projetos de Software Livre são testados? Como o conhecimento de teste é compartilhado entre os colaboradores do projeto? Qual o nível de formalismo usado para este conhecimento? Tais respostas são fornecidas com base na análise e compilação da literatura atual sobre o assunto.

Um dos grandes desafios que a Engenharia de Software enfrenta é: como construir *software* de boa qualidade com o menor custo, tempo e esforço possível? A literatura vem consolidando experiências e inovações resultantes do trabalho de muitos pesquisadores motivados em descrever formas mais eficazes de lidar com os problemas inerentes à construção de *software*.

Simultaneamente, uma proporção crescente de *software* vem sendo desenvolvida por grupos de indivíduos independentes, que trabalham de forma descentralizada e em muitos casos, geograficamente dispersos segundo uma filosofia que prega a

liberdade do conhecimento e da propriedade intelectual. O *software* que esses indivíduos, comumente chamados de colaboradores ou voluntários, desenvolvem pode ser livremente fornecido com o código fonte, utilizado, modificado e redistribuído por qualquer pessoa que se interessar.

Originalmente raros e reduzidos, estes grupos vêm estabelecendo-se gradualmente como organizações mais consolidadas, com nome próprio, equipe e missão bem definida: os projetos de *software* livre.

Vários sentidos e interpretações podem ser atribuídos ao *software* livre (do inglês, *Free Software*). No contexto deste trabalho, entende-se por *software* livre aquele que permite o uso, alteração e redistribuição por parte de seus usuários. Qualquer pessoa pode ler, estudar ou modificar livremente o código-fonte. Este sentido também é adotado pela *Free Software Foundation* (FSF)[1] e pela *Open Source Iniciative*[2], entidades importantes que foram criadas para divulgar e compartilhar informações sobre o *software* livre.

O desenvolvimento clássico de um *software* envolve algumas atividades básicas como análise, projeto, codificação e teste [27, 18]. No paradigma de *software* livre, é possível observar as mesmas atividades propostas pelo método clássico de engenharia de *software* porém com algumas particularidades. Várias trabalhos, como [17, 8, 4, 19, 22], abordam o processo de desenvolvimento de *software* livre com mais propriedade. Este trabalho se concentra na atividade de teste e para contextualizá-la melhor neste paradigma, dois projetos foram selecinados: Linux e LTP.

Em 1991, Linus Torvalds decidiu implementar o seu próprio núcleo (em inglês, *kernel*[3]) de sistema operacional e o batizou de Linux. Publicado na Internet no mesmo ano, o Linux rapidamente ganhou muitos usuários e muitos desenvolvedo-

[1] **www.fsf.org**

[2] **www.opensource.org**

[3] Nesta dissertação, optou-se por utilizar "*kernel*"como um termo genérico, parte comum a qualquer sistema operacional (sinônimo de núcleo).

res passaram a colaborar com o seu aprimoramento. Desde então, o Linux vem amadurecendo e não pára de crescer, demonstrando ser um campo fértil para a investigação da representação do conhecimento sobre a qualidade deste paradigma que é o *software* livre.

O LTP é um projeto de comunidade muito ativa voltado para o teste do Linux. Seu principal objetivo é resgatar o teste sistemático e contribuir com o crescimento da qualidade do Linux. O contexto desses dois projetos analisados de forma integrada resultam em uma combinação propícia a responder as questões de pesquisa levantadas por este trabalho.

Para tanto, o capítulo está organizado em quatro seções. Inicialmente, a Seção 2.1 discute o teste no contexto de *software* livre. A Seção 2.2 aborda o funcionamento básico do Linux e o seu desenvolvimento. A Seção 2.3 apresenta o LTP. O Linux e o LTP servem de exemplo para algumas questões de qualidade e teste levantadas por este Capítulo. Finalmente, a Seção 2.4 apresenta algumas considerações sobre teste e qualidade no paradigma de *software* livre.

2.1 Qualidade e Teste

Nos primeiros projetos de *software* livre, o teste como quesito para avaliar a qualidade não recebia uma atenção formal porque tudo era muito novo, os líderes dos projetos ainda estavam experimentando formas de organizar as cooperações e os desenvolvedores tinham como prioridade a codificação [30, 38, 39]. Com a explosão de novos projetos e cada vez mais pessoas interessadas em colaborar, o teste começou a ser inserido gradativamente. Hoje é possível observar grandes grupos interessados em consolidar o teste sistemático como ferramenta de garantia da qualidade em projetos de *software* livre.

A atividade de teste pode ser realizada basicamente de duas formas no contexto

do *software* livre. Quando o projeto é desenvolvido por uma pessoa ou um grupo muito pequeno, os próprios desenvolvedores e usuários ativos muitas vezes testam e reportam os erros encontrados. Já projetos maiores podem dispor de uma equipe para teste formada por membros internos do projeto, mas ainda assim, a participação dos usuários ativos que reportam erros é muito bem aceita e de grande importância para o aprimoramento do *software*.

Em projetos menores, que utilizam os usuários ativos para o teste, é comum o "teste não sistemático". Freqüentemente, os *bugs*[4], são descobertos por testes exploratórios e reportados em listas de discussão, fóruns ou lista de e-mails, ou seja, o usuário identifica falhas mediante a utilização do *software* e quanto mais familiaridade com o *software*, mais o usuário pode realizar operações que o ponha à prova.

Em projetos maiores, que dispõem de recursos para ter uma equipe de teste ou colaboradores que organizem grupos concentrados na qualidade do *software*, é mais comum encontrar "testes sistemáticos". Em geral, o teste segue métodos e utiliza técnicas. Existe, por exemplo, a preocupação de registrar os procedimentos de teste para que no futuro, sejam executados novamente. Há vários exemplos desta abordagem, como é o caso dos grupos de qualidade do Debian, Gnome, KDE e do Linux.

O grupo de garantia da qualidade da distribuição[5] Debian [11] convida os usuários a ajudarem a solucionar *bugs* e contribuírem com a documentação. A comunidade do ambiente gráfico GNOME voltada para a qualidade [12] também aguarda a colaboração dos usuários neste mesmo segmento. Já o grupo de garantia da qualidade do ambiente gráfico KDE [36] convida os usuários a participarem do aprimoramento

[4]Os *bugs* são incorretudes identificadas em um software

[5]As distribuições representam uma estratégia prática de popularização do Linux. Elas procuram agrupar programas como *drivers* e interfaces com o usuário de forma que o uso do Linux se torne cada vez mais amigável. Para maiores detalhes sobre as distribuições, consultar **www.linux.org/dist**

do *software* basicamente em quatro segmentos: teste (de versões estáveis, instáveis e *patches*[6]), documentações (com escrita de novas e revisão das existentes), comunicação e promoção (com escrita de artigos sobre *releases* atuais e futuras do *software*) e mantendo a documentação do desenvolvimento. Um outro exemplo é a comunidade do LTP [21]. O grupo se concentra principalmente no controle da qualidade do Linux e aguarda colaboração de desenvolvedores para implementação de testes. O projeto será discutido com mais detalhes na Seção 2.3.

Entre os milhares de projetos de *software* livre existentes, há um número de projetos que se destacam pela sua grande base de usuários, qualidade ou originalidade. As Seções 2.2 e 2.3 descreve dois projetos que se destacam por estes motivos. O primeiro é o Linux, um dos maiores e mais conhecidos projetos de *software* livre. O segundo é o LTP, projeto ativo que se apóia em técnicas, métodos e ferramentas de teste para propor avaliação da qualidade do Linux. A interseção entre esses dois projetos é objeto de estudo deste trabalho.

2.2 Funcionamento Básico do Linux

O sistema operacional é uma das partes de um "sistema computacional" responsável pelo uso e administração básica do *hardware*. Um sistema operacional a princípio inclui *kernel*, *drivers*, gerenciador de *boot*, interfaces com o usuário (linha de comando ou gráfica) e arquivos básicos.

O Linux, inicia sua participação no processo de inicialização (em inglês, *boot*) do sistema. A partir das instruções que são lidas no primeiro setor do disco rígido, o Linux aciona outros programas que dão suporte para a inicialização do sistema operacional.

O próximo passo é detectar os dispositivos de *hardware* essenciais do computador,

[6] *Patches* são pequenas atualizações de *software* que podem ser liberadas para os usuários com o intuito de corrigir alguma anomalia no *software*

como por exemplo, a placa de vídeo. Após detectar os dispositivos, as primeiras imagens do sistema são mostradas ao usuário. Em paralelo, o Linux verifica a memória e a prepara para o uso através de uma função de paginação bem como as interrupções, os discos, memória-*cache*, entre outros [20].

Em seguida, o sistema operacional está pronto para iniciar suas atividades. O Linux aciona as funções responsáveis em verificar quais *softwares* devem ser inicializados, por exemplo, o processo de *login* do usuário. Após o usuário efetuar o *login*, o Linux passa a executar funções, como a de controlar o uso da memória pelos programas ou a de atender chamadas de interrupção do *hardware*.

É interessante notar que as distribuições[7] executam o Linux com recursos e *drivers* básicos para o *hardware*, afinal carregar o suporte a todo tipo de dispositivo é algo inviável [16]. O Linux ficaria extremamente grande (ocuparia um espaço significativo na unidade de armazenamento) e somente os *drivers* relacionados ao *hardware* do computador em questão é que seriam usados. Para lidar com esse tipo de problema, os *drivers* são carregados como módulos após o Linux ser ativado. A questão é que carregar recursos por módulo gera uma queda de desempenho (pouco significativa em computadores rápidos) e por isso, muitos usuários preferem recompilar o Linux de seus sistemas para que esse carregue os *drivers* junto com sua inicialização, ou seja, sem usar módulos.

A árvore de desenvolvimento do Linux atende rigorosamente a uma hierarquia onde o Linus Torvalds se encontra no topo. É possível obter maiores informações sobre o desenvolvimento do Linux bem como o código-fonte em [10].

[7]Debian, Ubuntu, Suse entre outros são exemplos de distribuições que usam o Linux como *kernel*.

2.3 *Linux Test Project*

Existem atualmente diversos grupos dentro da comunidade de *software* livre envolvidos com a criação de testes para o Linux e com sua execução nos mais diversos tipos de *hardware*. O LTP é um desses projetos [13, 21]. Ele propõe avaliar a segurança, robustez e estabilidade do Linux e disponibilizar uma coleção de ferramentas para testá-lo. Alguns dos 3.000 casos de teste atualmente disponibilizados são utilizados para certificações de distribuições do Linux voltadas principalmente para utilização em computadores corporativos que disponiblizam algum tipo de serviço em rede. O LTP não provê documentação para todos os casos de teste já implementados e utilizados, o que torna trabalhoso o processo de desenvolvimento e manutenção dos mesmos, pois é necessário avaliar cada um para que se possa encontrar cenários não implementados, por exemplo.

Para obter um novo caso de teste é necessário ter o conhecimento sobre cada um dos existentes, identificando o processo de execução e quais parâmetros podem ser utilizados. Os casos de teste podem ser desenvolvidos dentro da própria comunidade ou podem ser desenvolvidos por empresas parceiras do projeto e depois disponibilizados para a comunidade.

O LTP mantém alguns *scripts* de teste, onde o mais utilizado é o *ltpstress.sh*, que possui um processo para seleção de novos casos de teste. O processo é constituído por três fases: a primeira fase consiste em avaliar o nível de recursos do sistema utilizados pelo caso de teste. A segunda fase consiste em medir a cobertura do caso de teste em relação ao sub-sistema ou funcionalidade do Linux a ser testada. Já a terceira, consiste em uma avaliação final do caso de teste, onde validam-se os resultados que podem ser obtidos em relação à métrica descrita para o mesmo e se a sua execução ocorreu como o esperado.

A PAN, também parte do LTP, é uma ferramenta utilizada para automatização

de execução e geração de relatórios. Um *script* de teste é carregado na PAN para iniciar a execução e, após ser carregado, é possível informar por exemplo a duração da execução ou a quantidade de iterações que o *script* deve ser executado. Ao término da execução, um relatório é salvo, o qual informa o resultado de execução de cada caso de teste individualmente. O relatório é resumido e informa *PASS*, caso o subsistema ou a funcionalidade tenha sido aprovada pelo caso de teste. Informa *FAIL* caso tenha acontecido alguma violação do caso de teste ou uma falha de execução do mesmo, tornando o relatório pouco informativo. Se as possíveis causas das falhas fossem informadas, o relatório se tornaria mais rico em informação e este é um grande desafio para a comunidade devido à não disponibilidade da documentação dos casos de teste.

2.4 Considerações Finais

É muito interessante observar que possa ter sucesso mundial um modelo de desenvolvimento aparentemente fundamentado no trabalho de voluntários, coordenados de maneira pouco formal e usando ferramentas extremamente simples. Mais surpreendente ainda é a forma em que esses colaboradores trocam informações e conhecimento dentro de suas comunidades virtuais (disponíveis na *internet*) e a excelência técnica dos desenvolvedores que participam do projeto.

O que se observou pela pesquisa bibliográfica sobre a percepção dos desenvolvedores acerca da qualidade é que ela é negligenciada sobre o seguinte aspecto: no início dos projetos há sempre uma preocupação maior com a implementação e atividades como teste sistemático e documentação passam a ter uma atenção maior com a evolução do projeto. Como o *software* livre é aberto, todos os artefatos que são gerados podem ser acompanhados pela comunidade. E esta sim, representa o termômetro da qualidade. É a comunidade que aprova ou reprova o andamento do

projeto.

Este sem dúvida foi um fator que motivou este trabalho a investigar este paradigma. O *software* livre viabiliza investigar como o conhecimento de teste é compartilhado entre os calaboradores do projeto e pela análise dos artefatos gerados, é possível identificar o nível de formalização usado e propor métodos que contribuam com a otimização do processo.

Neste paradigma podem ser encontrados muitos casos de sucesso com uma configuração interessante para a investigação da representação do conhecimento. O Linux foi escolhido por ser um projeto tradicional que consolida o processo de engenharia de *software* livre e o LTP por ser um projeto de aplicação prática, com comunidade ativa e que aguarda contribuições que potencializem o conhecimento associado ao processo de teste do Linux.

Capítulo 3

Representação de Conhecimento e Ontologias

Este Capítulo procura responder as seguintes questões: Quais os benefícios de uma representação formal? Uma vez o conhecimento representado, como é possível compartilhá-lo? Quais os benefícios de se utilizar Lógica de Descrição na representação do conhecimento? Quais os benefícios de se utilizar ontologias formais? Para tanto, o capítulo está organizado em quatro seções. Inicialmente, é apresentada na Seção 3.1 a noção de representação do conhecimento e terminologias associadas. Em seguida, a Seção 3.2 discute DL como formalismo lógico. A Seção 3.3 aborda as ontologias formais como uma forma de representação do conhecimento. Finalmente, a Seção 3.4 apresenta algumas considerações sobre o paradigma de representação formal.

3.1 Introdução

A Representação do Conhecimento [32, 5] investiga métodos para descrição do mundo real em alto nível, através de modelos conceituais, de forma que essas des-

crições possam ser efetivamente utilizadas para construir aplicações inteligentes. Neste contexto, aplicações inteligentes se referem à capacidade de um sistema encontrar conseqüências implícitas a partir de uma representação do conhecimento explícita.

As abordagens para representação do conhecimento em [3] podem ser divididas basicamente em duas categorias: formalismo baseado em lógica, cujo princípio fundamental é eliminar ambigüidades e capturar fatos do mundo real a princípio pela Lógica de Primeira Ordem ou por intermédio de suas variantes decidíveis; e a outra categoria é a de representações que não são baseadas em lógica, desenvolvidas para demonstrar noções cognitivas como é o caso de interfaces gráficas (mapas em estrutura de rede, entre outros).

O domínio discutido no Capítulo anterior nos mostra que o conhecimento sobre teste de *software* livre é mantido e registrado usando recursos pouco formais (por intermédio de lista de discussões muitas vezes contraditórias, listas de *e-mails*, fóruns, artefatos como código-fonte, entre outros). Isso motiva este trabalho a investigar a primeira abordagem, o formalismo baseado em lógica com o objetivo de agregar semântica ao domínio.

Sendo a Lógica de Primeira Ordem $\mathcal{L}$ indecidível, optou-se pelo estudo de um subconjunto decidível que é a Lógica de Descrição (em inglês, *description logic*) $\mathcal{DL}$. A lógica de descrição oferece muitas vantagens para a representação do conhecimento e o resultado prático das pesquisas está nas ferramentas desenvolvidas para a construção de aplicações baseadas no conhecimento. É o caso de ontologias que são aplicadas atualmente nas mais diversas áreas.

Na filosofia uma ontologia busca explanação sistemática da realidade, do que é ser. Quando olhamos para alguns problemas do mundo real pela ótica da ciência da computação, é possível chegar a conclusão que certos problemas não podem ser tratados exatamente como são, mas de uma maneira que os sistemas de informação

possam tratá-los eficientemente. Esta forma de interpretar os problemas do mundo real, despertou a investigação de ontologias como um artefato da engenharia do conhecimento. Freqüentemente essas ontologias representam teorias em lógica.

Na computação, quando uma ontologia faz uso de uma linguagem como OWL DL (*Web Ontology Language based on Description Logic*) [25], elas passam a ser ontologias formais constituídas por um vocabulário específico que descreve um modelo particular do mundo como classes, propriedades e instâncias.

3.2 DL como formalismo lógico

Lógica de Descrição (DL) é uma variante da Lógica de Primeira Ordem. Esta lógica envolve uma família de linguagens que podem ser usadas para representar o conhecimento de um universo de discurso. Basicamente, esta lógica descreve o mundo por intermédio de predicados unários (conceitos atômicos), predicados binários (regras atômicas), construtores e axiomas.

DL possui muitas características que fazem desta lógica um recurso poderoso para investigação de problemas ontológicos como, por exemplo, a definição de indivíduos, a natureza dos indivíduos, especialização de conceitos (relação é-um) e aspectos da relação todo-parte.

A primeira grande característica de DL está presente na forma de representar nichos de um domínio. Em uma base de conhecimento, é possível observar uma distinção clara entre o conhecimento sobre o domínio do problema e o conhecimento de um problema particular. De forma análoga, em uma base de conhecimento DL, existem dois componentes: a *TBox* e a *ABox*.

A TBox *(Terminological Box)*, contém o conhecimento do domínio na forma de terminologia, usada para construir declarações que descrevem propriedades gerais sobre os conceitos. Já a ABox *(Assertional Box)* contém o conhecimento específico

relacionado a indivíduos do domínio de discurso. Essa distinção permite especificar várias ABoxes para uma TBox, oferecendo a possibilidade de enriquecer a representação do domínio em questão. O Capítulo 5 e o Apêndice desta dissertação trazem exemplos de TBox e ABox especificadas na notação de DL.

Uma segunda característica de DL é o fato desta lógica ser usada como uma linguagem de modelagem já que tem uma semântica[1] intuitiva e uma sintaxe que se assemelha com a linguagem natural. Franz Baader destaca um exemplo desta característica na linguagem do cotidiano em [33]. "Um homem feliz é aquele casado com uma médica e que tem pelo menos cinco filhos professores". Este exemplo pode ser representado em DL fazendo uso de predicados unários, binários, construtores e axiomas como a seguir:

$$\texttt{HomemFeliz} \equiv \begin{pmatrix} \texttt{Humano} \sqcap \neg\, \texttt{Femea} \sqcap \exists\, \texttt{casadoCom.Medica} \sqcap \\ (\geq\ 5\ \texttt{temFilho}) \sqcap \forall\, \texttt{temFilho.Professor} \end{pmatrix}$$

HomemFeliz é um predicado unário, *casadoCom* é um predicado binário, $\sqcap, \neg, \exists, \geq$ e $\forall$ são exemplos de construtores e $\equiv$ é um exemplo de axioma.

A terceira grande característica de DL, que facilita a investigação de problemas ontológicos, refere-se a pesquisas conquistarem cada vez mais resultados teóricos e práticos como é o caso dos sistemas de representação do conhecimento baseados em DL oferecendo um arcabouço para novas pesquisas como é o caso deste trabalho.

Os algoritmos de raciocínio e suas complexidades têm influenciado no projeto de sistemas. Existem basicamente três abordagens para a implementação de sistemas de representação do conhecimento baseados em DL [3]. A primeira pode ser referenciada como "limitada porém completa", que inclui sistemas que são projetados para construções restritas onde, por exemplo, a relação é-um (em inglês, *subsumption*) pode ser computada eficientemente, possivelmente em tempo polinomial. A

[1] Tradicionalmente a semântica em DL é dada pelo modelo teórico Tarski-style.

segunda abordagem é referenciada como "expressiva porém incompleta", onde a idéia é prover uma linguagem expressiva mas em contra partida, o algoritmo de raciocínio é incompleto. A característica desta abordagem estimulou pesquisas que deram origem à terceira abordagem. Trata-se de uma abordagem "expressiva e completa"que não é tão eficiente como as abordagens anteriores, mas podem oferecer alguma testabilidade para a implementação de técnicas de raciocínio desenvolvidas por investigações teóricas.

O entendimento dessas abordagens é de fundamental importância para se tomar decisões de projeto. No caso deste trabalho e do domínio investigado, a primeira abordagem (limitada porém completa) está mais de acordo com a linha de pesquisa. Por intermédio desta abordagem é possível fazer uso da decidibilidade, onde as computações sobre a linguagem terminam num tempo finito e desta forma é possível realizar inferência automática, como classificação e verificação de consistência. A próxima Seção introduz com mais detalhes as ontologias que fazem uso de formalismos lógicos.

3.3 Ontologias Formais

Uma definição para ontologias formais é apresentada em [2] onde esta é definida como uma estrutura $\mathcal{O} := (C, \leq_C, R, \sigma, \leq_R)$. C e R são conjuntos disjuntos onde os membros de C são chamados de *conceitos* e os membros de R são chamados de *relações*. $\leq_C$ é uma ordem parcial de C chamada de *hierarquia de conceitos* ou *taxonomia* e $\leq_R$ é uma ordem parcial de R chamada de *hierarquia de relação*.

Se $c_1 \leq_C c_2$ para $c_1, c_2 \in C$, então c_1 é chamado de subconceito de c_2 e c_2 é superconceito de c_1. Obviamente a relação $\leq_C$ é supostamente conectada com conceitos que são definidos. Na literatura, as taxonomias são construídas usando-se relação de subconjuntos, ou seja, $c_i \leq_C c_j$ se e somente se para todo $o \in c_i$ tem-se

$o \in c_j$. Esta definição de $\leq_C$ produz uma ordem parcial em C que é usada pelas ontologias formais.

Os conceitos podem ser vistos como coleções de objetos que podem ter certas características instanciadas. Neste trabalho, para uma ontologia $\mathcal{O}$ existe um conjunto de características $\mathcal{F} = f_1, \ldots, f_n$ e para cada característica f_i existe um domínio $D_i = v_{il}, \ldots, v_{im_i}$ que define os possíveis valores e características. Então, um objeto $o = ([f_1 = v_1], \ldots, [f_n = v_n])$ é caracterizado por valores para cada característica (muitas vezes uma característica é o nome de identificação de um objeto que por sua vez tem uma única combinação de características).

As ontologias formais oferecem vários benefícios para a representação do conhecimento. O primeiro deles é que métodos formais introduzirem o conceito de corretude ao sistema. O segundo benefício é o fato da estrutura lógica de um problema ser enfatizada ao invés de uma construção engessada de um procedimento para resolvê-lo. Isso permite que o conhecimento seja reusado e inferido sobre o domínio.

3.4 Considerações Finais

Com os recursos fornecidos pelas ciências contemporâneas, a representação do conhecimento se tornou muito mais dinâmica. Sob a ótica da Ciência da Computação é possível encontrar o paradigma formal fazendo uso de linguagens capazes de representar sintática e semanticamente o conhecimento, tornando-o processável por máquinas e viabilizando a troca de informação entre sistemas computacionais.

É no paradigma formal que este trabalho se concentra. Ele foi escolhido como paradoxo ao domínio de *software* livre, que já faz uso de muitos métodos informais. O principal objetivo é agregar os benefícios de um método formal a um domínio que aguarda contribuições que possam otimizar a qualidade de *software* livre.

A Lógica de Descrição foi escolhida principalmente pelo seu arcabouço científico,

pelos sistemas de representação do conhecimento baseados em DL, como ambientes de desenvolvimento (facilitando o processo de implementação), raciocinadores (que auxiliam no processo de verificação e validação dos sistemas), pela sua sintaxe e semântica (semelhante à linguagem natural porém sem ambigüidades).

Ontologias formais foram escolhidas principalmente por fazerem uso de formalismo lógico, o que permite a utilização de DL como linguagem de representação dos questionamentos ontológicos sobre o domínio discutido no Capítulo anterior.

Parte II

Descrição Metodológica

Capítulo 4

Instrumental de Pesquisa

O objetivo deste capítulo é apresentar o instrumental de pesquisa adotado para o desenvolvimento deste projeto bem como introduzir nomenclaturas utilizadas. A Seção 4.1 apresenta uma contextualização do instrumental de pesquisa. Os problemas levantados por este projeto bem como a delimitação do escopo para uma melhor investigação. Os métodos, técnicas e ferramentas são discutidos na Seção 4.2. A Seção 4.3 apresenta a nomenclatura utilizada na Parte III desta dissertação. A Seção 4.4 apresenta algumas considerações finais.

4.1 Escopo do Projeto

O Linux vem evoluindo e amadurecendo gradativamente, conquistando cada vez mais usuários pela sua qualidade. Apesar de diversas funcionalidades e melhorias desenvolvidas no Linux, os critérios de elaboração de teste sistemático, definição de um vocabulário formal para o domínio bem como o registro eficiente do conhecimento dos projetistas de teste sobre suas decisões de cobertura do Linux, ainda são motivo de longas discussões na comunidade.

Com o intuito de oferecer uma contribuição que reduza estas dificuldades, este

trabalho levantou a hipótese de utilizar DL como formalismo lógico para representar parte do conhecimento associado ao teste do Linux, bem como a definição de um vocabulário do domínio de teste por intermédio de ontologias formais.

Como o Linux dispõe de vários subsistemas, a investigação de todos eles não seria possível devido a escassez de recursos e a premência do tempo. Este fator de completude exigiu que a pesquisa fosse delimitada quanto a sua extensão já que não é possivel abranger todo o âmbito de teste de todos os subsistemas do Linux.

Neste caso, utilizou-se o método da amostragem intencional, que consiste em obter um juízo sobre o total (universo), mediante a compilação e exame de apenas uma parte, a amostra selecionada. Dentre os subsistemas do Linux, foi selecionado o *Virtual File System* (VFS) por ser um subsistema de fundamental importância e pelo fato de o LTP conter vários casos de teste para este subsistema. Só que ainda assim, o VFS dispõe de quatro objetos para serem investigados (*inode, superblock, dentry* e *file*), discutidos no Capítulo A, exigindo mais uma vez a delimitação do campo de investigação. O objeto *file* foi selecionado por ter um tamanho ideal, compatível com o tempo disponível para a investigação.

Para tal, uma representação formal do conhecimento apoiada por uma ontologia pode agregar muitos benefícios ao processo de desenvolvimento do Linux que se encontra contínuamente em evolução. Três benefícios importantes podem ser destacados. O primeiro é prover um vocabulário formal do domínio de teste do Linux que represente o consenso de um grupo. O segundo é o registro semântico dos critérios de elaboração dos testes sistemáticos. O terceiro é o registro semântico do conhecimento dos projetistas de teste. Os benefícios citados estão alinhados com as atuais necessidades da comunidade de teste do Linux que aguardam por contribuições que otimizem este processo.

A análise inicial do domínio de teste do Linux foi feita por intermédio de pesquisas bibliográficas e reunião com especialistas. Por intermédio da elaboração de

um diagrama do modelo conceitual[1] em UML (*Unified Modeling Language*)[18] foi possível visualizar a presença de dois domínios distintos, o de teste de *software* e o de sistema operacional. Por uma decisão de projeto, optou-se por desenvolver uma ontologia para cada um dos domínios identificados, permitindo desta forma modularizá-los. A ontologia OSOnto (*Operating System Ontology*) se concentra no domínio de sistema operacional e a ontologia SwTO (*Software Testing Ontology*) se concentra no domínio de teste de *software*. Porém, esses domínios podem se integrar, o que deu origem à SwTO^I (*Software Test Ontology Integrated*), uma ontologia que se concentra no domínio de teste de sistema operacional. Desta forma, o conhecimento sobre o teste do Linux forma uma possível ABox para a SwTO^I, discutida no próximo Capítulo.

Nas próximas seções serão discutidos métodos, técnicas, ferramentas e convenções de formato comuns as três ontologias citadas. As nomenclaturas utilizadas serão discutidas a fim de tornar mais claro o entendimento do Capítulo 5 e do Apêndice A.

4.2 Métodos, Técnicas e Ferramentas Utilizados

Com os problemas apontados no início deste Capítulo e com o escopo definido para o projeto, foi selecionado o método de Uschold e Gruninger [37] que define passos para a construção de ontologias. A escolha deste método deveu-se à sua adequação à natureza deste trabalho de pesquisa, visto tratar-se de um projeto com um ciclo de vida mais curto que os projetos de maior envergadura. O projeto contou com a participação do grupo GIALix (Grupo de Inteligência Atificial para aprimoramento do Linux), porém foi praticamente executado por apenas uma pessoa.

[1] Em UML, um modelo conceitual é exibido como um conjunto de diagramas de estrutura estática usado para enfatizar conceitos, atributos dos conceitos e associações, e não entidades de *software*.

O método adotado é composto por quatro fases referentes ao desenvolvimento de uma ontologia: (i) identificação do objetivo e escopo, (ii) construção, (iii) avaliação e (iv) a documentação.

A *primeira fase* consistiu no entendimento preliminar do domínio. Foram realizadas pesquisas documentais e reuniões com especialistas do domínio. Posteriormente foram construídos vários diagramas representando o modelo conceitual em UML que por sua vez foram discutidos e refinados em reuniões com os especialistas.

A elaboração de diagramas do modelo conceitual na fase preliminar explicitou a presença de dois domínios distintos porém relacionados. O domínio de sistemas operacionais e o domínio de teste de *software* o que deu origem, a princípio, a duas ontologias nomeadas de OSOnto (*Operating System Ontology*), descrita no Apêndice e a SwTO (*Software Test Ontology*).

Os objetivos ontológicos foram definidos bem como o escopo de cada ontologia por meio da formulação de questões de competência em linguagem natural.

A *segunda fase* iniciou com a investigação de ontologias existentes para os domínios identificados a fim de reusar o conhecimento. Para o domínio de sistema operacional a ontologia mais significativa encontrada foi a BLO (*Basic Linux Ontology*)[2] que incluem conceitos de um curso introdutório de Linux da Universidade de Leeds [9, 23]. Já no domínio de teste de *software*, a ontologia mais significativa foi a SWEBOK (*Software Engineering Body of Knowledge*) proposta por um grande comitê técnico da IEEE, procura fornecer definições e terminologia para as várias áreas de conhecimento da engenharia de *software* e dentre elas está o teste de *software* [1, 34].

Da ontologia formal BLO, foi possível extrair conhecimento e reusá-lo na OSOnto na forma de classes, propriedades e instâncias. Já da ontologia informal SWEBOK, foi possível extrair o conhecimento do texto em linguagem natural e transcrevê-lo

[2] A *Basic Linux Ontology* foi construída por um especialista do domínio utilizando a linguagem OWL-Lite e o editor Protégé. Esta ontologia é parte de um projeto maior chamado SWALE.

para a linguagem OWL DL, dando origem a uma ontologia formal, a SwTO.

Ainda na segunda fase, a atividade de captura dos conceitos incluiu a identificação, descrição e escolha de nomes dos conceitos associados ao domínio de sistema operacional e teste de *software*, assim como a identificação dos relacionamentos entre eles.

A atividade de codificação das ontologias incluiu a escolha da linguagem de implementação, OWL DL [25], e das ferramentas utilizadas, no caso foram, o editor de ontologias Protégé [28], versão 3.3.1, e os motores de inferência RACER (*Renamed ABox and Concept Expression Reasoner Professional*) Pro [29], versão acadêmica 1.9.0 como raciocinador primário e Pellet [26], versão 1.4, como raciocinador secundário.

OWL DL foi a linguagem utilizada para implementação das ontologias pelas seguintes razões:

- Contém as primitivas para a representação dos conceitos, relacionamentos e axiomas da ontologia. Além disso, estas primitivas oferecem a expressividade necessária, tais como a representação de classes disjuntas e a definição de classes a partir da combinação de outras classes;
- Permite inferência lógica realizada por mecanismo de raciocínio automático que implementa um algoritmo correto e completo para raciocínio com classes;
- Conta com uma comunidade de pesquisa atuante nos meios acadêmico e comercial;
- Permite a utilização da ontologia desenvolvida em ambientes Web;
- Pode ser utilizada através de um editor de ontologias, no caso, o Protégé [28].

O editor Protégé, por sua vez, foi escolhido como ambiente de desenvolvimento pelas seguintes razões:

- É de uso livre e amplamente utilizado;
- Oferece material de apoio de muito boa qualidade, tal como tutoriais e listas de discussão na *Internet*;
- Possui diversos *plug-ins*. O PROMPT *Tab* é um *plug-in* que viabiliza o gerenciamento de várias ontologias no Protégé, oferecendo a comparação de versões, extração de partes específicas de uma ontologia ou a mesclagem de duas ontologias em uma só. Já o TGVizTab e o Jambalaya são *plug-ins* de visualização[3] de ontologias.

A escolha do Racer como motor de inferência primário deveu-se à facilidade de uso juntamente com o Protégé e às características próprias do raciocinador, tais como rapidez de execução, a utilização de um algoritmo de inferência completo e correto, e a disponibilidade de diversas formas de interface, como a interface especificada pelo Grupo de Implementação de Lógica Descritiva (DIG - *Description Logic Implementation Group*). O Pellet, em determinada fase do projeto, se mostrou instável, apresentando erros ilegíveis em sua execução. Por ter o código-fonte aberto, diferentemente do Racer, o Pellet permite investigar a fundo o que pode estar causando o erro no processo de verificação de consistência da ontologia. Porém, esse tipo de investigação demanda tempo e esforço. Diante disto, o Pellet foi mantido no projeto como raciocinador secundário.

Ambos os raciocinadores assumem a premissa lógica de mundo aberto (*open world assumption*), segundo a qual o que não pode ser provado como verdadeiro não é considerado falso e sim desconhecido (*nil*).

Ainda na segunda fase, foi feita a integração entre as ontologias, dando origem à "SwTO *Integrated*", ou simplesmente SwTO^I, que comporta um conjunto de axiomas, propriedades e instâncias pertinentes à integração de OSOnto e SwTO. A

[3] O GVizTab e o Jambalaya foram utilizados para gerar os gráficos presentes no Capítulo 5 e no Apêndice deste trabalho

Seção 4.3 apresenta a nomenclatura adotada para apresentação das ontologias que foi adotada no do Capítulo 5 e no Apêndice.

A avaliação da ontologia SwTO^I foi realizada na *terceira fase* do desenvolvimento, que consistiu na verificação das características da ontologia quanto a sua estrutura, consistência, completude e concisão. O método estatístico de avaliação baseado na estrutura da ontologia, proposto em [15], é baseado na teoria de grafos e foi utilizado com o objetivo de complementar o método de avaliação de Uschold e Gruninger [37]. O resultado desta fase é apresentado no Capítulo 7.

As atividades propostas na *quarta fase*, a documentação, foi executada ao longo das fases descritas acima.

Algumas técnicas de programação foram adotadas na implementação das ontologias, tais como:

- No nome das classes e propriedades, adotou-se o padrão de concatenação de palavras com iniciais maiúsculas, sem o uso de conectores como hífen ou ponto. O singular também foi adotado como padrão para os nomes dos conceitos.

- A definição do nome das propriedades, também foi padronizada. Salvo exceções, os nomes de propriedades são prefixados com verbos como *has* (tem) e *is* (é), de maneira a associar uma propriedade à sua inversa.

- Foram utilizados axiomas de fechamento (*closure axioms*) na descrição de classes, por meio de restrição de propriedade, sempre que aplicável. Detalhamentos deste tipo de axioma podem ser encontrados ao longo do Capítulo 5 e no Apêndice.

- Foram utilizados axiomas de cobertura (*covering axioms*) na descrição de classes sempre que aplicável. Esta técnica consiste em definir que todos os membros da superclasse são membros de uma de suas subclasses, ou seja, não existe

nenhuma instância da superclasse que não seja também instância de uma de suas subclasses. Detalhamentos deste tipo de axioma podem ser encontrados ao longo do Capítulo 5 e no Apêndice.

- O motor de inferência RACER foi utilizado para fazer a verificação de consistência da ontologia durante toda a atividade de codificação e o Pellet em parte da codificação, de forma a identificar rapidamene a inserção de definições inconsistentes.

Optou-se, de modo geral, pela não definição de domínios e contradomínios para as propriedades, visto que tal especificação, devido à premissa de mundo aberto, funciona como um axioma e não como uma restrição para inferência sobre a ontologia. Entretanto, algumas propriedades dispõem de domínio e contradomínio para facilitar a instanciação da Ontologia. A figura 4.1 ilustra a propriedade objeto *isKernelOf* cujo domínio é a classe *Kernel* e o contra domínio é a classe *OperatingSystem*. Ao in-

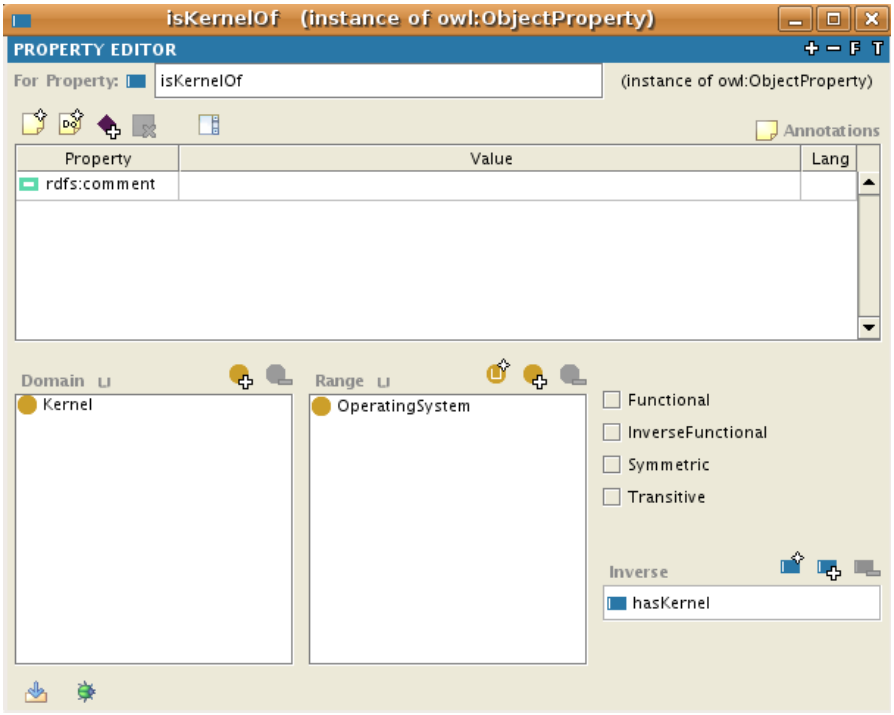

Figura 4.1: Domínio e contradomínio da propriedade *isKernelOf*

serir a instância *Linux* para a classe *Kernel* e ao associar essa instância com outras, através da propriedade *isKernelOf*, o editor Protégé automaticamente apresenta as possíveis instâncias com as quais *Linux* pode ser associado com base no contra-domínio da propriedade *isKernelOf*. A figura 4.2 ilustra este processo de associação entre instâncias através da propriedade objeto *isKernelOf*. Observe que dentre as instâncias da classe *OperatingSystem*, a instâcia *Linux* se relaciona com apenas três, Ubuntu, Debian e Suse. Nestes casos em que o domínio e contradomínio foram informados, não ocorreram inferências inesperadas por parte do raciocinador.

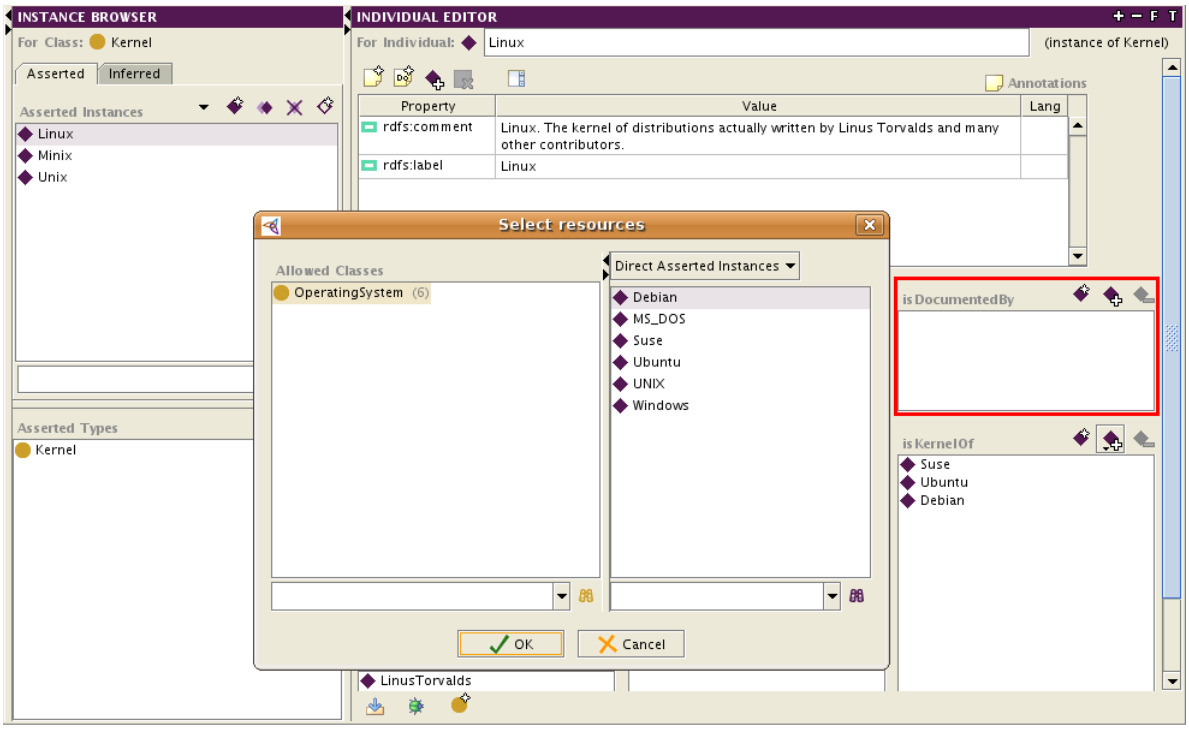

Figura 4.2: Associação entre instâncias através da propriedade *isKernelOf*

O projeto também contou com a gerência de configuração feita por intermédio do Subversion (SVN)[4]. Esta ferramenta foi utilizada para controlar os vários ciclos de desenvolvimento das ontologias, as versões do código-fonte geradas em cada ciclo e toda a documentação promovendo uma abordagem colaborativa ao desenvolvimento. O SVN permitiu uma cooperação mais ágil entre os participantes do projeto. Com

[4]http://subversion.tigris.org

o código-fonte e a documentação disponível para os membros do grupo GIALix, no qual este trabalho esteve inserido, acomapanhavam de forma descentralizada o desenvolvimento das ontologias, revisavam a documentação e contribuiam com sugestões para o projeto.

A vantagem da abordagem colaborativa de desenvolvimento está em aumentar as chances da ontologia ser aceita como um padrão para um determinado domínio, pois possui a contribuição de um conjunto de pessoas para a sua formação. Por outro lado a ontologia resultante desse processo pode demorar a atingir a maturidade esperada devido o ponto de vista de mais de um indivíduo.

4.3 Convenções de Formato

Esta seção define a nomenclatura adotada na apresentação das ontologias SwTO^I (ver Capítulo 5) e OSOnto (ver Apêndice A). A língua inglesa foi utilizada na definição dos nomes e comentários das classes e propriedades, de forma que a ontologia possa ser discutida e utilizada em meios internacionais como a *internet*.

As palavras em inglês estão destacadas em *"itálico"*. Ao longo dos Capítulos 5 e Apêndice A, componentes ontológicos, tais como classes e propriedades, nomeados em inglês, estão destacados em ***"negrito itálico"***.

Quanto ao modo de apresentação das ontologias, cabem as seguintes observações:

- O nível de granularidade[5] escolhido para apresentar as ontologias foi as classes nomeadas.

- A partir das classes nomeadas, foram detalhados todos os outros conceitos relacionados como propriedades, instâncias e condições lógicas associadas (como disjunções).

[5]No contexto desta dissertação, entende-se por granularidade o ato ou efeito de particionar algo, sinônimo de detalhamento.

- Os formalismos adotados para representar os conceitos das ontologias são: lógica de descrição, fragmentos do código OWL DL. Também foram utilizados diagramas gerados pelos *plugins* Jambalaya e TGVizTab presentes no Protégé.

- Cada classe nomeada está descrita em linguagem natural, seguida por sua descrição formal que compõe a TBox da Ontologia, onde são definidas condições sobre a interpretação da classe, discutidas a seguir nas Definições de *Condição Necessária*, *Condição Suficiente* e *Condição Necessária e Suficiente*.

- As propriedades em OWL podem assumir características (tais como funcionalidade, funcionalidade inversa, transitividade e simetria), bem como domínio e contradomínio específicos. Tais características específicas assumidas pelas propriedades objeto, que relacionam um indivíduo a outro indivíduo, denotadas como P_O e propriedades de tipo de dados, que relaciona um indivíduo a um valor de tipo de dados *XML Schema*, denotadas como P_D, foram informadas sempre que aplicável.

- Finalmente para complementar, foram apresentadas algumas instâncias do domínio somente para algumas classes, dentre elas, as mais significativas. Estas instâncias fazem parte da Abox da ontologia e compõem o conhecimento representado.

As condições sobre a interpretação das classes podem ser: necessárias, suficientes ou necessárias e suficientes.

Definição 4.1 (Condição Necessária) *Ser $\mathcal{A}$ é condição necessária para $\mathcal{X}$ ser $\mathcal{F}$ quando é impossível que $\mathcal{X}$ seja $\mathcal{F}$ e não seja $\mathcal{A}$.*

Exemplo: ser um assento ($\mathcal{A}$) é uma condição necessária para algo ($\mathcal{X}$) ser uma cadeira ($\mathcal{F}$).

Se imaginarmos que assento e cadeira são classes, X pertence a classe cadeira somente se for um assento. Nem todo assento é uma cadeira.

Definição 4.2 (Condição Suficiente) *Ser $\mathcal{B}$ é condição suficiente para $\mathcal{X}$ ser $\mathcal{F}$ quando é impossível que $\mathcal{X}$ seja $\mathcal{B}$ e não seja $\mathcal{F}$.*

Exemplo: ser humano ($\mathcal{B}$) é uma condição suficiente para algo ($\mathcal{X}$) ser um animal ($\mathcal{F}$).

Se imaginarmos que humano e animal são classes, o fato de X pertencer a classe humano garante que X é animal.

Definição 4.3 (Condição Necessária e Suficiente) *Ser $\mathcal{C}$ é condição necessária e suficiente para $\mathcal{X}$ ser $\mathcal{F}$ quando é impossível que $\mathcal{X}$ seja $\mathcal{F}$ sem ser $\mathcal{C}$ e que seja $\mathcal{C}$ sem ser $\mathcal{F}$.*

Exemplo: ser formado por moléculas compostas de dois átomos de hidrogênio e um átomo de oxigênio ($\mathcal{C}$) é uma condição necessária e suficiente para algo ($\mathcal{X}$) ser água ($\mathcal{F}$).

Isso quer dizer que moléculas compostas de dois átomos de hidrogênio e um átomo de oxigênio é equivalente a água.

4.4 Considerações Finais

Este capítulo apresentou o instrumental metodológico utilizado para o desenvolvimento deste trabalho. O objeto *file* foi escolhido como amostra intencional de investigação porque se mostrou compatível com o cronograma deste projeto, por ser um objeto relevante dentro do VFS do Linux e pelo fato do LTP conter vários casos de teste para este objeto, indicando oportunidade e viabilidade de pesquisa. Os métodos, técnicas e ferramentas utilizados neste trabalho bem como as convenções de formato são de fundamental importância para a compreensão da Parte III desta dissertação.

No Capítulo seguinte a ontologia SwTO^I será apresentada. Componentes como classes, propriedades, instâncias e condições lógicas do domínio de teste do Linux serão discutidos.

Parte III

Apresentação, Análise e Interpretação de Resultados

Capítulo 5

SwTO^I: uma Ontologia com Aplicação em Teste do Linux

O objetivo deste Capítulo é apresentar a SwTO^I (*Software Test Ontology Integrated*). Para tanto, foi adotada a seguinte organização: a Seção 5.1 apresenta uma contextualização do domínio representado. O conhecimento formal da ontologia é descrito na Seção 5.2. A seção 5.3 apresenta algumas considarações finais sobre o Capítulo.

5.1 Introdução

Como foi discutido no Capítulo 4, durante a fase de identificação do objetivo e escopo da ontologia observou-se a presença de dois domínios, o de teste de *software* e o de sistema operacional, para os quais foram desenvolvidas respectivamente duas ontologias: a SwTO (*Software Test Ontology*) e a OSOnto (*Operating Systems Ontology*). Esse tratamento modular dos domínios permite o uso individual das ontologias em contextos diferentes. Por exemplo, aplicações centradas no processo de teste podem reusar o conhecimento específico da SwTO enquanto a implementação de ferramentas que têm relação com o domínio de sistema operacional podem reusar o conhecimento da OSOnto sem necessariamente levar em consideração a SwTO. Entretanto, se o objetivo for o reuso dos dois domínios de forma integrada, temos a

SwTO^I ou SwTO *Integrated*.

Esta ontologia contém a SwTO e mais algumas características pertinentes à integração dos domínios que ocorre pela importação da OSOnto[1]. A Figura 5.1 apresenta a SwTO como um conjunto contido na SwTO^I que por sua vez tem uma relação (chamada $owl : imports$) com o conjunto OSOnto. Este processo de importação é

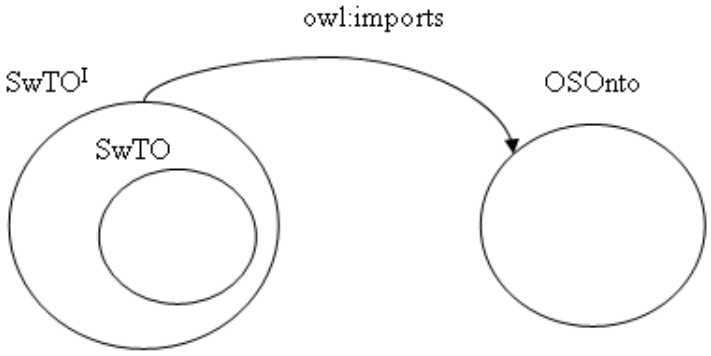

Figura 5.1: A SwTO^I e a sua relação com a SwTO e OSOnto

realizado através do editor Protégé. Com a SwTO^I em edição, é possível importar o código-fonte *OSOnto.owl* adicionando a seguinte linha ao código-fonte da SwTO^I:

$$< owl : importsrdf : resource = "http : //www.owl - ontologies.com/OSOnto.owl" / >$$

A partir deste momento, a OSOnto fica disponível somente para leitura e não pode ser editada. Já a SwTO^I pode ser editada normalmente. Esta caracteristica da ontologia disponível para edição pode ser configurada. Se a OSOnto for configurada como a ontologia ativa, a SwTO^I fica disponível somente para leitura.

No decorrer deste Capítulo, sempre que algum recurso (tais como classes e propriedades) da ontologia OSOnto for referenciado, este será precedido por um identificador (em inglês, *namespace*), que consiste de uma declaração da linguagem OWL usada para diferenciar os recursos de cada ontologia em caso de importação.

[1]Para maiores detalhes sobre a ontologia OSOnto, consultar o Apêndice A.

5.2 A SwTO^I

Seguindo a nomenclatura definida no Capítulo 4, a SwTO^I é formalizada por sua TBox e ABox. Primeiramente, são definidas condições para a interpretação das classes. Em seguida são detalhadas as propriedades que têm relação com a classe em questão e suas disjunções com outras classes.

As propriedades em OWL podem ter características (tais como funcionalidade, funcionalidade inversa, transitividade e simetria), bem como domínio e contradomínio específicos. No decorrer da discussão sobre as propriedades que descrevem às classes, as características específicas assumidas pelas propriedades serão informadas. Cabe ressaltar que todas as vezes que uma classe tem subclasses, estas por sua vez herdam as propriedades (P_O e P_D) e condições[2] que foram definidas para a superclasse. E finalmente, para complementar, algumas instâncias do domínio são apresentadas. A seguir, cada item se refere a uma classe da SwTO^I.

- ***SoftwareTestDomainConcept:*** são subclasses desta classe, todos os conceitos que fazem parte ou que têm alguma relação com o domínio de teste de *software*.
  SoftwareTestDomainConcept é a classe de mais alto nível na ontologia e é subclasse de owl:Thing.

  $$\texttt{SoftwareTestDomainConcept} \sqsubseteq \top$$

  A Figura 5.2 gerada pelo *plug-in* Jambalaya introduz as subclasses de *SoftwareTestDomainConcept*.

- **SoftwareTestProcess:** o teste é uma verificação dinâmica do *software*, realizado para um conjunto finito de casos de teste, adequadamente selecionados de um domínio. Os conceitos, estratégias, técnicas e mensuração do teste precisam ser integrados em um processo bem definido e controlado, de forma que pessoas possam executá-lo. O resultado deste processo deve ser comparado ao comportamento esperado para o *software*. O processo de teste de *software* guia as atividades que devem ser realizadas pela equipe de teste. Todas essas atividades são devidamente planejadas de forma que a qualidade do *software* possa ser garantida.
  Condição Necessária:

  (i) *SoftwareTestProcess* é subclasse de *SoftwareTestDomainConcept* por ser um conceito

[2]Para obter maiores informações sobre os tipo de condições referenciadas neste trabalho, consultar a Seção 4.3.

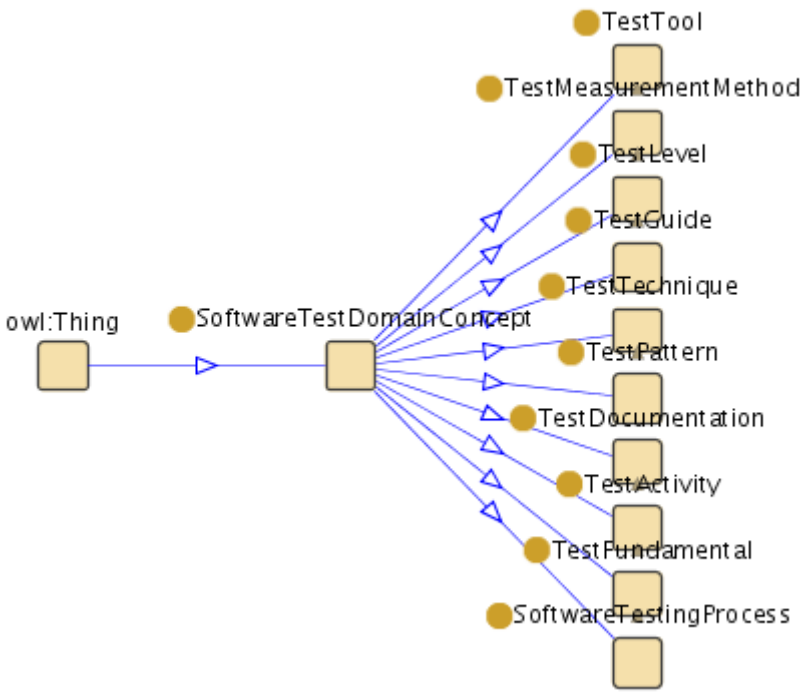

Figura 5.2: *SoftwareTestDomainConcept* e suas subclasses

que faz parte do domínio.

$$\texttt{SoftwareTestProcess} \sqsubseteq \texttt{SoftwareTestDomainConcept}$$

(ii) Um processo de teste possui várias atividades. A expressão a seguir define que indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *hasTestActivity* com indivíduos da classe *TestActivity*.

$$\texttt{SoftwareTestProcess} \sqsubseteq \forall\, \texttt{hasTestActivity} \,.\, \texttt{TestActivity}$$

A Figura 5.3 ilustra essa restrição. *SoftwareTestProcess* é subclasse da classe anônima que contém indivíduos que só se relacionam, via propriedade `hasTestActivity`, com indivíduos da classe *TestActivity*.

(iii) Indivíduos da classe *SoftwareTestProcess* relacionam-se através da propriedade *hasActivity* com pelo menos um indivíduo da classe *TestActivity*.

$$\texttt{SoftwareTestProcess} \sqsubseteq \exists\, \texttt{hasTestActivity} \,.\, \texttt{TestActivity}$$

A Figura 5.4 ilustra essa restrição. *SoftwareTestProcess* é subclasse da classe anônima que contém indivíduos que se relacionam, via propriedade `hasTestActivity`, com pelo menos um indivíduo da classe *TestActivity*. Os indivíduos da classe anônima podem se relacionar pela mesma propriedade com outros indivíduos que não sejam da classe *TestActivity* (destacado na Figura 5.4 por linhas tracejadas). Entretanto, quando combinamos a restrição universal com a existencial na descrição de uma classe é possível inferir que os indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *hasActivity* com indivíduos da classe *TestActivity* e com pelo menos um indivíduo desta mesma classe.

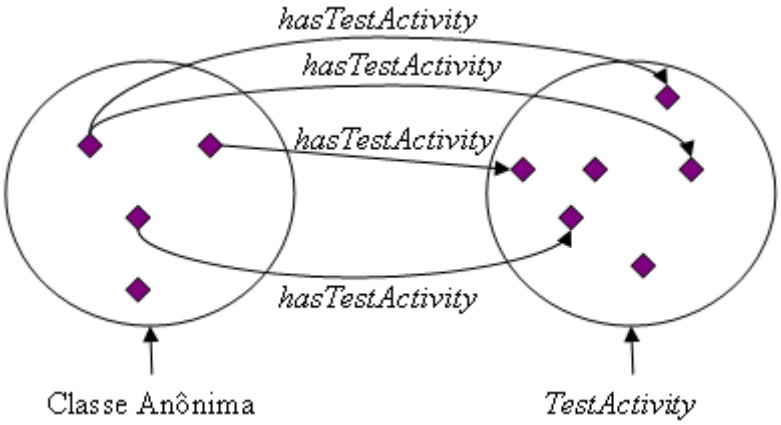

Figura 5.3: Um esquema da restrição universal ($\forall$ *hasTestActivity* . *TestActivity*)

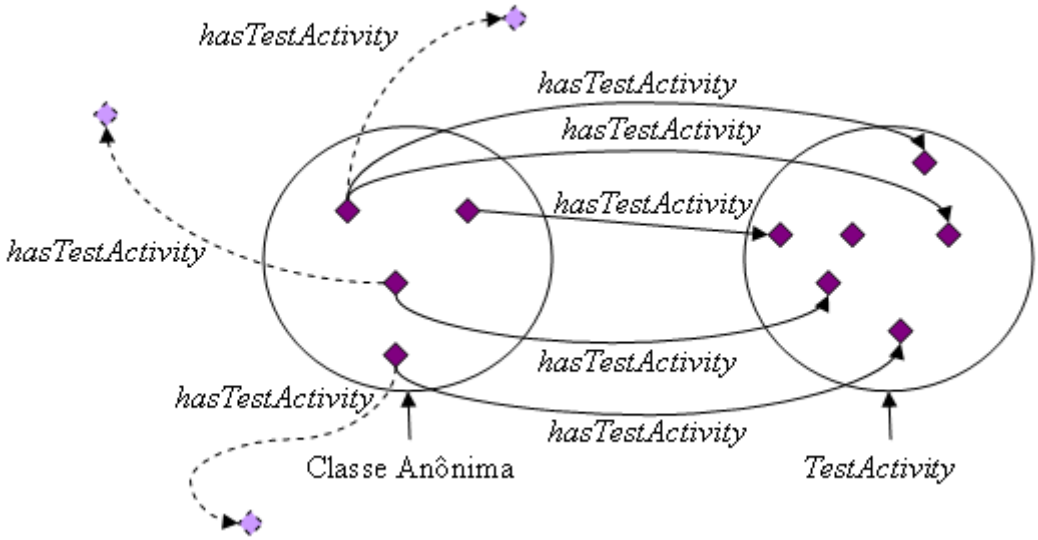

Figura 5.4: Um esquema da restrição existencial ($\exists$ *hasTestActivity* . *TestActivity*)

(iv) Indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *hasTestDocumentation* com indivíduos da classe *TestDocumentation*.

$$\texttt{SoftwareTestProcess} \sqsubseteq \forall\, \texttt{hasTestDocumentation} \,.\, \texttt{TestDocumentation}$$

(v) Os processos de teste podem se apoiar em guias de teste. Desta forma, indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *hasTestGuide* com indivíduos da classe *TestGuide*.

$$\texttt{SoftwareTestProcess} \sqsubseteq \forall\, \texttt{hasTestGuide} \,.\, \texttt{TestGuide}$$

(vi) Indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *hasTestFundamental* com indivíduos da classe *TestFundamental*.

$$\texttt{SoftwareTestProcess} \sqsubseteq \forall\, \texttt{hasTestFundamental} \,.\, \texttt{TestFundamental}$$

(vii) Indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *hasTestLevel* com indivíduos da classe *TestLevel*.

$$\texttt{SoftwareTestProcess} \sqsubseteq \forall\, \texttt{hasTestLevel} \,.\, \texttt{TestLevel}$$

(viii) Os processos de teste podem comportar padrões de teste semelhantes aos padrões de projeto (*design patterns*), cuja finalidade é facilitar o reuso. Desta forma, indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *hasTestPattern* com indivíduos da classe *TestPattern*.

$$\texttt{SoftwareTestProcess} \sqsubseteq \forall\, \texttt{hasTestPattern} \,.\, \texttt{TestPattern}$$

(ix) Indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *hasTestMeasurementMethod* com indivíduos da classe *TestMeasurementMethod*.

$$\texttt{SoftwareTestProcess} \sqsubseteq \forall\, \texttt{hasTestMeasurementMethod} \,.\, \texttt{TestMeasurementMethod}$$

(x) Indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *hasTestTechnique* com indivíduos da classe *TestTechnique*.

$$\texttt{SoftwareTestProcess} \sqsubseteq \forall\, \texttt{hasTestTechnique} \,.\, \texttt{TestTechnique}$$

(xi) Indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *hasTestTool* com indivíduos da classe *TestTool*.

$$\texttt{SoftwareTestProcess} \sqsubseteq \forall\, \texttt{hasTestTool} \,.\, \texttt{TestTool}$$

(xii) Indivíduos da classe *SoftwareTestProcess* só relacionam-se através da propriedade *isSoftwareTestProcessOf* com indivíduos da classe *Software* da OSOnto.

$$\texttt{SoftwareTestProcess} \sqsubseteq \begin{pmatrix} \forall\, \texttt{isSoftwareTestProcessOf} \,. \\ \textbf{OSOnto:Software} \end{pmatrix}$$

(xiii) Indivíduos da classe *SoftwareTestProcess* relacionam-se através da propriedade *isSoftwareTestProcessOf* com pelo menos um indivíduo da classe *Software* da OSOnto.

$$\texttt{SoftwareTestProcess} \sqsubseteq \begin{pmatrix} \exists\, \texttt{isSoftwareTestProcessOf} \,. \\ \textbf{OSOnto:Software} \end{pmatrix}$$

Propriedades - *Sobre as propriedades P_O que descrevem a classe SoftwareTestProcess, pode-se dizer que as nove propriedades descritas a seguir são subpropriedades de hasPart que por sua vez é inversa de isPartOf. Ambas são transitivas. Esta hierarquia de propriedades objeto pode ser observada pela Figura 5.5*

(i) A propriedade *hasTestActivity* é inversa de *isTestActivityOf*. O domínio desta propriedade é a classe *SoftwareTestProcess* e o contradomínio é *TestActivity*.

$$\texttt{hasTestActivity} \equiv \texttt{isTestActivityOf}^{-}$$
$$\top \sqsubseteq \forall\, \texttt{hasTestActivity} \,.\, \texttt{SoftwareTestProcess}\ (\texttt{SoftwareTestProcess} \neq \bot)$$
$$\top \sqsubseteq \forall\, \texttt{hasTestActivity}^{-} \,.\, \texttt{TestActivity}\ (\texttt{TestActivity} \neq \bot)$$

(ii) A propriedade *hasTestDocumentation* é inversa de *isTestDocumentationOf*. O domínio desta propriedade é a classe *SoftwareTestProcess* e o contradomínio é *TestDocumen-*

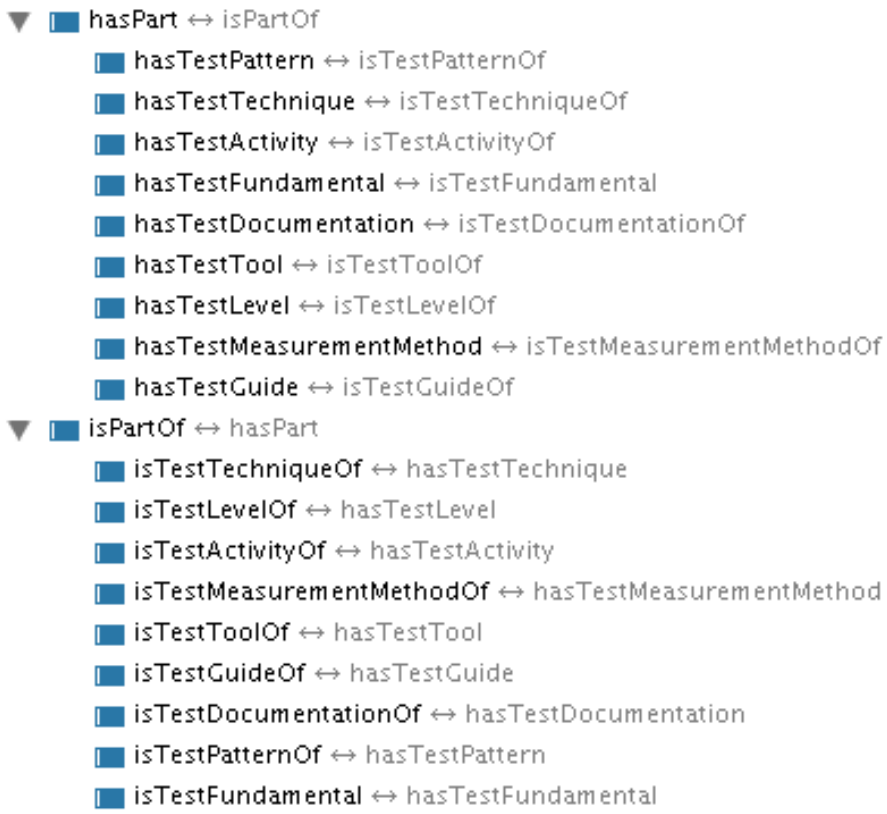

Figura 5.5: Hierarquia das propriedades *hasPart* e *isPartOf*

tation.

$$\texttt{hasTestDocumentation} \equiv \texttt{isTestDocumentationOf}^{-}$$
$$\top \sqsubseteq \forall\, \texttt{hasTestDocumentation} \,.\, \texttt{SoftwareTestProcess}\ (\texttt{SoftwareTestProcess} \neq \bot)$$
$$\top \sqsubseteq \forall\, \texttt{hasTestDocumentation}^{-} \,.\, \texttt{TestDocumentation}\ (\texttt{TestDocumentation} \neq \bot)$$

(iii) A propriedade *hasTestGuide* é inversa de *isTestGuideOf*. O domínio desta propriedade é a classe *SoftwareTestProcess* e o contradomínio é *TestGuide*.

$$\texttt{hasTestGuide} \equiv \texttt{isTestGuideOf}^{-}$$
$$\top \sqsubseteq \forall\, \texttt{hasTestGuide} \,.\, \texttt{SoftwareTestProcess}\ (\texttt{SoftwareTestProcess} \neq \bot)$$
$$\top \sqsubseteq \forall\, \texttt{hasTestGuide}^{-} \,.\, \texttt{TestGuide}\ (\texttt{TestGuide} \neq \bot)$$

(iv) A propriedade *hasTestFundamental* é inversa de *isTestFundamentalOf*. O domínio desta propriedade é a classe *SoftwareTestProcess* e o contradomínio é *TestFundamental*.

$$\texttt{hasTestFundamental} \equiv \texttt{isTestFundamentalOf}^{-}$$
$$\top \sqsubseteq \forall\, \texttt{hasTestFundamental} \,.\, \texttt{SoftwareTestProcess}\ (\texttt{SoftwareTestProcess} \neq \bot)$$
$$\top \sqsubseteq \forall\, \texttt{hasTestFundamental}^{-} \,.\, \texttt{TestFundamental}\ (\texttt{TestFundamental} \neq \bot)$$

(v) A propriedade *hasTestLevel* é inversa de *isTestLevelOf*. O domínio desta propriedade

é a classe *SoftwareTestProcess* e o contradomínio é *TestLevel*.

$$\texttt{hasTestLevel} \equiv \texttt{isTestLevelOf}^-$$
$$\top \sqsubseteq \forall\, \texttt{hasTestLevel} \,.\, \texttt{SoftwareTestProcess}\ (\texttt{SoftwareTestProcess} \neq \bot)$$
$$\top \sqsubseteq \forall\, \texttt{hasTestLevel}^- \,.\, \texttt{TestLevel}\ (\texttt{TestLevel} \neq \bot)$$

(vi) A propriedade *hasTestPattern* é inversa de *isTestPatternOf*. O domínio desta propriedade é a classe *SoftwareTestProcess* e o contradomínio é *TestPattern*.

$$\texttt{hasTestPattern} \equiv \texttt{isTestPatternOf}^-$$
$$\top \sqsubseteq \forall\, \texttt{hasTestPattern} \,.\, \texttt{SoftwareTestProcess}\ (\texttt{SoftwareTestProcess} \neq \bot)$$
$$\top \sqsubseteq \forall\, \texttt{hasTestPattern}^- \,.\, \texttt{TestPattern}\ (\texttt{TestPattern} \neq \bot)$$

(vii) A propriedade *hasTestMeasurementMethod* é inversa de *isTestMeasurementMethodOf*. O domínio desta propriedade é a classe *SoftwareTestProcess* e o contradomínio é *TestMeasurementMethod*.

$$\texttt{hasTestMeasurementMethod} \equiv \texttt{isTestMeasurementMethodOf}^-$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasTestMeasurementMethod} \,.\, \texttt{SoftwareTestProcess} \\ (\texttt{SoftwareTestProcess} \neq \bot) \end{pmatrix}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasTestMeasurementMethod}^- \,.\, \texttt{TestMeasurementMethod} \\ (\texttt{TestMeasurementMethod} \neq \bot) \end{pmatrix}$$

(viii) A propriedade *hasTestTechnique* é inversa de *isTestTechniqueOf*. O domínio desta propriedade é a classe *SoftwareTestProcess* e o contradomínio é *TestTechnique*.

$$\texttt{hasTestTechnique} \equiv \texttt{isTestTechniqueOf}^-$$
$$\top \sqsubseteq \forall\, \texttt{hasTestTechnique} \,.\, \texttt{SoftwareTestProcess}\ (\texttt{SoftwareTestProcess} \neq \bot)$$
$$\top \sqsubseteq \forall\, \texttt{hasTestTechnique}^- \,.\, \texttt{TestTechnique}\ (\texttt{TestTechnique} \neq \bot)$$

(ix) A propriedade *hasTestTool* é inversa de *isTestToolUsedIn*. O domínio desta propriedade é a classe *SoftwareTestProcess* e o contradomínio é *TestTool*.

$$\texttt{hasTestTool} \equiv \texttt{isTestToolUsedIn}^-$$
$$\top \sqsubseteq \forall\, \texttt{hasTestTool} \,.\, \texttt{SoftwareTestProcess}\ (\texttt{SoftwareTestProcess} \neq \bot)$$
$$\top \sqsubseteq \forall\, \texttt{hasTestTool}^- \,.\, \texttt{TestTool}\ (\texttt{TestTool} \neq \bot)$$

(x) A propriedade *isSoftwareTestProcessOf* tem como domínio a classe *SoftwareTestProcess* e como contradomínio a classe *OSOnto:Software*.

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isSoftwareTestProcessOf} \,.\, \texttt{SoftwareTestProcess} \\ (\texttt{SoftwareTestProcess} \neq \bot) \end{pmatrix}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isSoftwareTestProcessOf}^- \,.\, \textbf{OSOnto} : \texttt{Software} \\ (\texttt{OSOnto} : \texttt{Software} \neq \bot) \end{pmatrix}$$

Disjunções - *Sobre as classes disjuntas de SoftwareTestProcess, pode-se dizer:*

(i) A classe *SoftwareTestProcess* é disjunta de *TestLevel*, *TestDocumentation*, *TestTool*, *TestFundamental*, *TestActivity*, *TestTechnique*, *TestGuide*, *TestMeasurementMethods*

e *TestPattern* já que não compartilha instâncias com as mesmas.

$$\begin{aligned}\texttt{SoftwareTestProcess} \sqsubseteq\ & \neg\ \texttt{TestLevel} \sqcap \\ & \neg\ \texttt{TestDocumentation} \sqcap \\ & \neg\ \texttt{TestTool} \sqcap \\ & \neg\ \texttt{TestFundamental} \sqcap \\ & \neg\ \texttt{TestActivity} \sqcap \\ & \neg\ \texttt{TestTechnique} \sqcap \\ & \neg\ \texttt{TestGuide} \sqcap \\ & \neg\ \texttt{TestMeasurementMethods} \sqcap \\ & \neg\ \texttt{TestPattern}\end{aligned}$$

Instanciação - *Sobre possíveis instâncias da classe SoftwareTestProcess, pode-se dizer:*

$$\texttt{SoftwareTestProcess}(\texttt{LinuxTestProcess})$$

A seguir é possível observar a relação da instância com as propriedades P_O.

$$\begin{aligned}\texttt{hasTestActivity}(\texttt{LinuxTestProcess}, \texttt{KernelPartialTestPlanning}) \\ \texttt{hasTestDocumentation}(\texttt{LinuxTestProcess}, \texttt{inode\_1}) \\ \texttt{hasTestFundamental}(\texttt{LinuxTestProcess}, \texttt{testForDefectIdentification\_1}) \\ \texttt{hasTestLevel}(\texttt{LinuxTestProcess}, \texttt{performanceTest\_1}) \\ \texttt{hasTestTechnique}(\texttt{LinuxTestProcess}, \texttt{mutationTesting\_1}) \\ \texttt{hasTestTool}(\texttt{LinuxTestProcess}, \texttt{TeSG}) \\ \texttt{isSoftwareTestProcessOf}(\texttt{LinuxTestProcess}, \texttt{Linux})\end{aligned}$$

- ***TestActivity:*** esta classe representa as atividades macro que podem ser realizadas dentro de um ciclo de teste de *software*.
  Condição Necessária:

  (i) *TestActivity* é subclasse de *SoftwareTestDomainConcept*.

  $$\texttt{TestActivity} \sqsubseteq \texttt{SoftwareTestDomainConcept}$$

  (ii) A atividade de teste pode ser realizada pela equipe de teste (formada por pessoas) ou por *software*. Desta forma, indivíduos da classe *TestActivity* só relacionam-se através da propriedade *isPerformedBy* com indivíduos que estejam presentes na união das classes *TestTeam* e *OSOnto:Software*.

  $$\texttt{TestActivity} \sqsubseteq\ \forall\ \texttt{isPerformedBy}(\texttt{TestTeam} \sqcup \texttt{OSOnto} : \texttt{Software})$$

  (iii) Indivíduos da classe *TestActivity* relacionam-se através da propriedade *isPerformedBy* com pelo menos um indivíduo presente na união entre as classes *TestTeam* e *OSOnto:Software*.

  $$\texttt{TestActivity} \sqsubseteq\ \exists\ \texttt{isPerformedBy}(\texttt{TestTeam} \sqcup \texttt{OSOnto} : \texttt{Software})$$

  (iv) Indivíduos da classe *TestActivity* só relacionam-se através da propriedade *isTestActivityOf* com indivíduos da classe *SoftwareTestProcess* e com pelo menos um indivíduo desta classe através da mesma propriedade.

  $$\begin{aligned}\texttt{TestActivity} &\sqsubseteq\ \forall\ \texttt{isTestActivityOf} . \texttt{SoftwareTestProcess} \\ \texttt{TestActivity} &\sqsubseteq\ \exists\ \texttt{isTestActivityOf} . \texttt{SoftwareTestProcess}\end{aligned}$$

Disjunções - *Sobre as classes disjuntas de TestActivity, pode-se dizer:*

(i) A classe *TestActivity* é disjunta das classes *TestLevel*, *TestDocumentation*, *TestTool*, *TestFundamental*, *SoftwareTestProcess*, *TestTechnique*, *TestGuide*, *TestMeasurementMethods* e *TestPattern* já que não compartilha instâncias com as mesmas.

$$\begin{aligned} \texttt{TestActivity} \sqsubseteq\ & \neg\ \texttt{TestLevel} \sqcap \\ & \neg\ \texttt{TestDocumentation} \sqcap \\ & \neg\ \texttt{TestTool} \sqcap \\ & \neg\ \texttt{TestFundamental} \sqcap \\ & \neg\ \texttt{SoftwareTestProcess} \sqcap \\ & \neg\ \texttt{TestTechnique} \sqcap \\ & \neg\ \texttt{TestGuide} \sqcap \\ & \neg\ \texttt{TestMeasurementMethods} \sqcap \\ & \neg\ \texttt{TestPattern} \end{aligned}$$

A Figura 5.6 introduz as subclasses de *TestActivity* que serão detalhadas a seguir. Todas elas são disjuntas entre si.

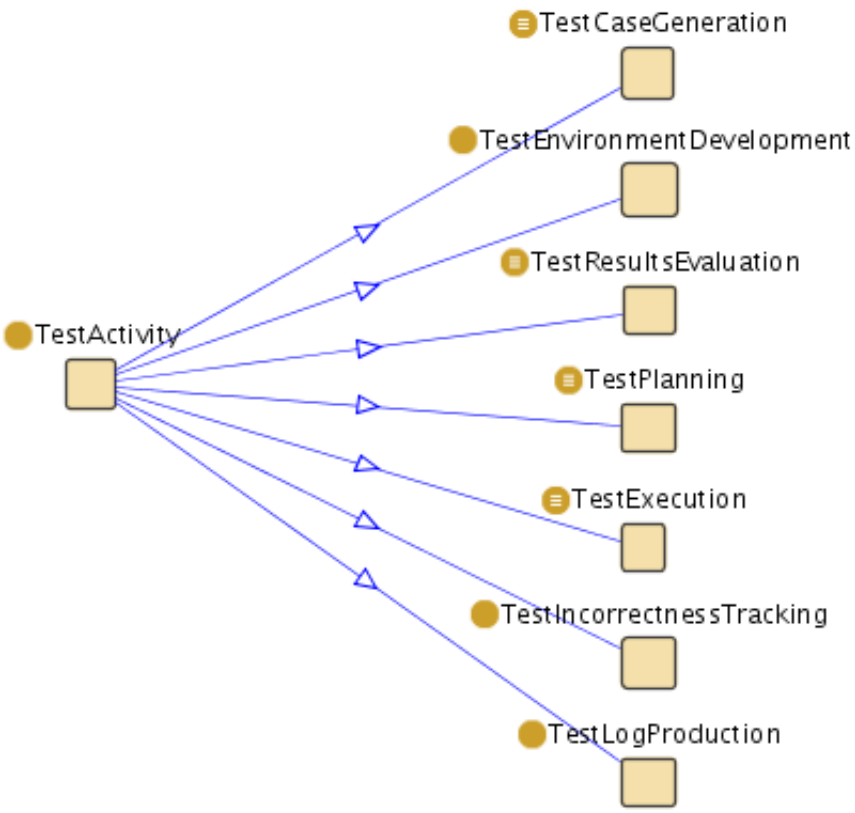

Figura 5.6: *TestActivity* e suas subclasses

- ***TestIncorrectnessTracking:*** esta classe representa a atividade de registro das incorretudes de um software. Quando os *"bugs"* são devidamente documentados, desde o motivo do seu nascimento até a sua morte, é possível

extrair uma experiência muito valiosa deste processo. Pela trajetória dos *bugs* é possível identificar quando eles foram introduzidos no *software* e o que desencadeou essa inserção (como por exemplo, uma definição pobre dos requisitos, declaração incorreta de uma variável, alocação inadequada de memória, erro de programação, entre outros). A análise do registro gerado por esta atividade contribui inclusive para melhorar aspectos da engenharia de *software*.
Condição Necessária:

(i) *TestIncorrectnessTracking* é subclasse de *TestActivity*.

$$\texttt{TestIncorrectnessTracking} \sqsubseteq \texttt{TestActivity}$$

(ii) Esta atividade registra as incorretudes referentes a um *software*. Indivíduos da classe *TestIncorrectnessTracking* só relacionam-se através da propriedade *recordsIncorrectness* com indivíduos da classe *Incorrectness*

$$\texttt{TestIncorrectnessTracking} \sqsubseteq \forall\, \texttt{recordsIncorrectness} \,.\, \texttt{Incorrectness}$$

(iii) A atividade de registro de defeito visa apontar pelo menos uma incorretude no *software*. Indivíduos da classe *TestIncorrectnessTracking* relacionam-se através da propriedade *recordsIncorrectness* com pelo menos um indivíduo da classe *Incorrectness*.

$$\texttt{TestIncorrectnessTracking} \sqsubseteq \exists\, \texttt{recordsIncorrectness} \,.\, \texttt{Incorrectness}$$

(iv) A atividade de registro de defeito sucede a atividade de avaliação do resultado de teste. Indivíduos da classe *TestIncorrectnessTracking* só relacionam-se através da propriedade *succeedsActivity* com indivíduos da classe *TestResultEvaluation*.

$$\texttt{TestIncorrectnessTracking} \sqsubseteq \forall\, \texttt{succeedsActivity} \,.\, \texttt{TestResultEvaluation}$$

Propriedades - *Sobre as propriedades que definem a classe TestIncorrectnessTracking, pode-se dizer:*

(i) *recordsIncorrectness* e *succeedsActivity* são propriedades objeto. *succeedsActivity* é transitiva e inversa de *antecedesActivity*.
Para exemplificar a transitividade de *succeedsActivity*, digamos que se uma determinada atividade de registro de defeito c sucede uma determinada atividade de avaliação dos resultados do teste b que por sua vez sucede um determinado planejamento a, por transitividade, é possível inferir que a atividade c sucede a atividade de planejamento a, realizada antes de c.

$$\texttt{recordsIncorrectness} \in \texttt{P}_\texttt{0} \tag{5.1}$$

$$\texttt{succeedsActivity} \in \texttt{P}_\texttt{0} \tag{5.2}$$

$$\texttt{succeedsActivity} \equiv \texttt{antecedesActivity}^- \tag{5.3}$$

$$\texttt{succeedsActivity} \in \mathbf{R}_+ \tag{5.4}$$

- ***TestExecution:*** esta classe representa a atividade de execução de um teste. Esta atividade exige registros claros e bem documentados dos passos seguidos, ou seja, dos casos de teste. Isso se deve ao fato de permitir que outras pessoas reproduzam os mesmos experimentos e possam averiguar os resultados obtidos.
Condição Necessária e Suficiente:

(i) Algo executa casos de teste para o qual foi especificado pelo menos um valor esperado

se e somente se trata-se de uma atividade de execução.

$$\texttt{TestExecution} \equiv \forall \begin{pmatrix} \texttt{executesTestCase}(\texttt{TestCase} \sqcap \\ (\exists\ \texttt{hasValue}\ .\ \texttt{TestCaseExpectedValue})) \end{pmatrix}$$

De forma mais detalhada, essa condição pode ser entendida como: indivíduos da primeira classe anônima só relacionam-se através da propriedade *executesTestCase* com indivíduos presentes na interseção da classe *TestCase* com uma segunda classe anônima que contém indivíduos que relacionam-se através da propriedade *hasValue* com pelo menos um indivíduo da classe *TestCaseExpectedValue*.

A Figura 5.7 apresenta um esquema que ilustra essa condição. As linhas tracejadas indicam que os indivíduos podem se relacionar pela mesma propriedade com indivíduos de outras classes.

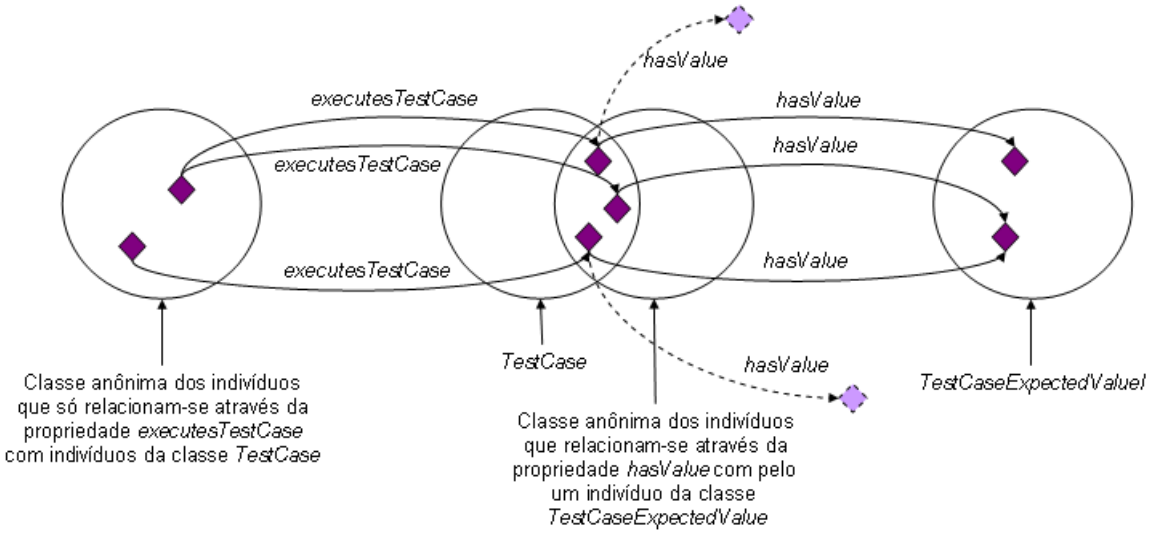

Figura 5.7: Esquema da condição necessária e suficiente da classe *TestExecution*

Condição Necessária:

(i) *TestExecution* é subclasse de *TestActivity*.

$$\texttt{TestExecution} \sqsubseteq \texttt{TestActivity}$$

(ii) A atividade de execução antecede a avaliação do resultado de teste. Indivíduos da classe *TestExecution* só relacionam-se através da propriedade *antecedesActivity* com indivíduos da classe *TestResultEvaluation*.

$$\texttt{TestExecution} \sqsubseteq \forall\ \texttt{antecedesActivity}\ .\ \texttt{TestResultEvaluation}$$

(iii) A atividade de execução pode suceder a atividade de planejamento ou a atividade de geração de casos de teste. Em alguns casos o planejamento não prevê a geração e sim o reuso e logo em seguida a execução dos casos de teste. Por este motivo, consideramos a união entre as classes *TestPlanning* e *TestCaseGeneration*.

$$\texttt{TestExecution} \sqsubseteq \forall\ \texttt{succeedsActivity}\ .\ (\texttt{TestPlanning} \sqcup \texttt{TestCaseGeneration})$$

Propriedades - *Sobre as propriedades que participam da descrição da classe TestExecution, pode-se dizer:*

(i) *antecedesActivity* é uma propriedade objeto inversa de *succeedsActivity*.

$$\begin{aligned} \texttt{antecedesActivity} &\in \texttt{P}_\texttt{O} \\ \texttt{antecedesActivity} &\equiv \texttt{succeedsActivity}^- \end{aligned}$$

(ii) *executesTestCase* é uma propriedade objeto cujo domínio é a classe *TestExecution*.

$$\begin{aligned} \texttt{executesTestCase} &\in \texttt{P}_\texttt{O} \\ \top &\sqsubseteq \forall\, \texttt{executesTestCase} \,.\, \texttt{TestExecution}\ (\texttt{TestExecution} \neq \bot) \end{aligned}$$

- ***TestPlanning:*** Assim como qualquer outra atividade de um projeto, o teste e suas atividades devem ser planejados. Os aspectos do planejamento de um teste, incluem coordenção de pessoal, gerenciamento da complexidade do teste e planejamento para possíveis resultados indecidíveis, por exemplo, com o crescimento das variáveis de entrada o teste pode não ser realizado em tempo polinomial. O planejamento para o teste inicia logo no processo de especificação de requisitos, e os procedimentos podem ser desenvolvidos de forma sistemática e contínua.
Condição Necessária e Suficiente:

(i) Uma atividade de planejamento registra as decisões tomadas em um plano de teste. Desta forma, indivíduos são inferidos como instâncias da classe *TestPlanning* se e somente se relacionam-se através da propriedade *createsTestPlan* com indivíduos da classe *TestPlan* e com pelo menos um indivíduo desta mesma classe através da propriedade *createsTestPlan*.

$$\begin{aligned} \texttt{TestPlanning} &\equiv \forall\, \texttt{createsTestPlan} \,.\, \texttt{TestPlan} \\ \texttt{TestPlanning} &\equiv \exists\, \texttt{createsTestPlan} \,.\, \texttt{TestPlan} \end{aligned}$$

Condição Necessária:

(i) *TestPlanning* é subclasse de *TestActivity*.

$$\texttt{TestPlanning} \sqsubseteq \texttt{TestActivity}$$

(ii) As atividade de planejamento antecede o desenvolvimento do ambiente de teste, a geração de casos de teste, a execução dos testes ou a avaliação dos resultados. Desta forma, indivíduos da classe *TestPlanning* só relacionam-se através da propriedade *antecedesActivity* com indivíduos da classe união entre *TestEnvironmentDevelopment*, *TestCaseGeneration*, *TestExecution* e *TestResultEvaluation*.

$$\texttt{TestPlanning} \sqsubseteq \forall\, \texttt{antecedesActivity} \,.\, \left(\begin{array}{c} \texttt{TestEnvironmentDevelopment} \sqcup \\ \texttt{TestCaseGeneration} \sqcup \\ \texttt{TestExecution} \sqcup \\ \texttt{TestResultEvaluation} \end{array} \right)$$

(iii) Em um processo de teste as atividades de execução e avaliação são indispensáveis. Logo, indivíduos da classe *TestPlanning* relacionam-se através da propriedade *antecedesActivity* com pelo menos um indivíduo da classe *TestExecution* e relacionam-se

através da mesma propriedade com pelo menos um indivíduo da classe *TestResultEvaluation*.

$$\texttt{TestPlanning} \sqsubseteq \exists\, \texttt{antecedesActivity} \,.\, \texttt{TestExecution}$$
$$\texttt{TestPlanning} \sqsubseteq \exists\, \texttt{antecedesActivity} \,.\, \texttt{TestResultEvaluation}$$

A Figura 5.8 ilustra a classe *TestPlanning* e uma classe resultante da união entre *TestEnvironmentDevelopment, TestCaseGeneration, TestExecution* e *TestResultEvaluation*. Digamos que dentro da classe "união", existam quatro elementos. O triângulo *isósceles* é um indivíduo da classe *TestExecution*, o triângulo *retângulo* é um indivíduo da classe *TestResultEvaluation*, o *paralelogramo* é um indivíduo da classe *TestCaseGeneration* e o *retângulo* é um indivíduo da classe *TestEnvironmentDevelopment*. Com base nas restrições definidas no item (ii) e (iii), cada elemento da classe *TestPlanning* só relaciona-se através da propriedade *antecedesActivity* com indivíduos da classe união e cada indivíduo da classe *TestPlanning* relaciona-se através da mesma propriedade com pelo menos um indivíduo da classe *TestExecution* e com pelo menos um indivíduo da classe *TestResultEvaluation*.

Observe que o primeiro elemento da classe *TestPlanning* só relaciona-se com os indivídos da classe *TestExecution* e *TestResultEvaluation*. O segundo indivíduo relaciona-se com todos os indivíduos da classe união.

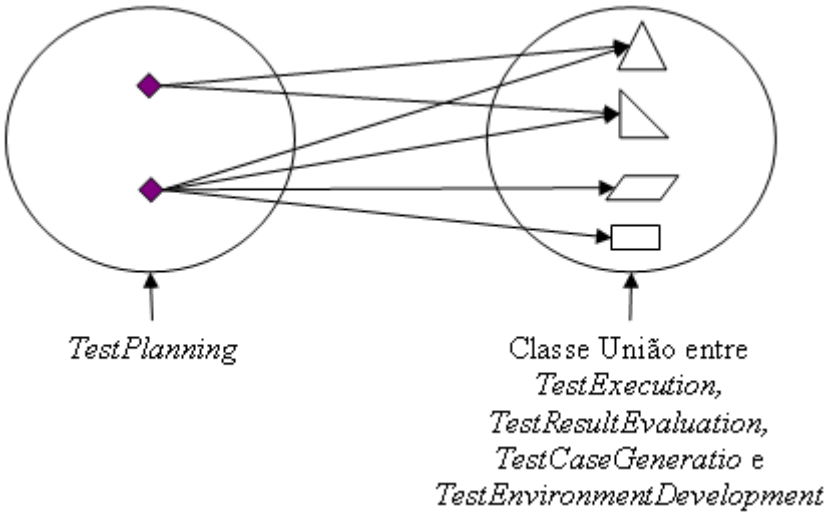

Figura 5.8: Representação da condição necessária da classe *TestPlanning*

- ***TestCaseGeneration:*** esta classe representa a atividade de geração de casos de teste que freqüentemente se baseia na estratégia de teste que se pretende realizar e/ou em alguma técnica de teste.
  Condição Necessária e Suficiente:

  (i) Um indivíduo é inferido como uma atividade de geração de caso de teste se e somente se gerar casos de teste e no mínimo um caso de teste.

$$\texttt{TestCaseGeneration} \equiv \forall\, \texttt{generatesTestCase} \,.\, \texttt{TestCase}$$
$$\texttt{TestCaseGeneration} \equiv \geq\, \texttt{1}\ \texttt{generatesTestCase} \,.\, \texttt{TestCase}$$

  Condição Necessária:

(i) *TestCaseGeneration* é subclasse de *TestActivity*.

$$\texttt{TestCaseGeneration} \sqsubseteq \texttt{TestActivity}$$

(ii) A atividade de geração de casos de teste antecede a atividade de execução. Desta forma, indivíduos da classe *TestCaseGeneration* só relacionam-se através da propriedade *antecedesActivity* com indivíduos da classe *TestExecution*.

$$\texttt{TestCaseGeneration} \sqsubseteq \forall\, \texttt{antecedesActivity}\, .\, \texttt{TestExecution}$$

(iii) A atividade de geração se baseia em pelo menos um nível de teste ou técnica de teste. Desta forma, indivíduos da classe *TestCaseGeneration* relacionam-se através da propriedade *isBaseOn* com pelo menos um indivíduo da classe união entre *TestLevel* e *TestTechnique*.

$$\texttt{TestCaseGeneration} \sqsubseteq \exists\, \texttt{isBasedOn}\, (\texttt{TestLevel} \sqcup \texttt{TestTechnique})$$

(iv) A atividade de geração sucede as atividades de planejamento ou desenvolvimento do ambiente de teste. Desta forma, indivíduos da classe *TestCaseGeneration* só relacionam-se através da propriedade *succeedsActivity* com indivíduos da classe união entre *TestPlanning* e *TestEnvironmentDevelopment*.

$$\texttt{TestCaseGeneration} \sqsubseteq \begin{pmatrix} \forall\, \texttt{succeedsActivity}\, (\texttt{TestPlanning} \\ \sqcup\, \texttt{TestEnvironmentDevelopment}) \end{pmatrix}$$

- ***TestEnvironmentDevelopment:*** o desenvolvimento do ambiente de teste procura selecionar ferramentas que auxiliem na elaboração dos testes.
  Condição Necessária:

  (i) *TestEnvironmentDevelopment* é subclasse de *TestActivity*.

  $$\texttt{TestEnvironmentDevelopment} \sqsubseteq \texttt{TestActivity}$$

  (ii) A atividade de desenvolvimento do ambiente de teste antecede a atividade de execução. Indivíduos da classe *TestEnvironmentDevelopment* só relacionam-se através da propriedade *antecedesActivity* com indivíduos da classe *TestExecution*.

  $$\texttt{TestEnvironmentDevelopment} \sqsubseteq \forall\, \texttt{antecedesActivity}\, .\, \texttt{TestExecution}$$

  (iii) A atividade de desenvolvimento do ambiente de teste sucede a atividade de planejamento. Indivíduos da classe *TestEnvironmentDevelopment* só relacionam-se através da propriedade *succeedsActivity* com indivíduos da classe *TestPlanning*.

  $$\texttt{TestEnvironmentDevelopment} \sqsubseteq \forall\, \texttt{succeedsActivity}\, .\, \texttt{TestPlanning}$$

  (iv) A atividade de desenvolvimento do ambiente de teste seleciona ferramentas que facilitem a engenharia do teste. Indivíduos da classe *TestEnvironmentDevelopment* só relacionam-se através da propriedade *selectsTestTool* com indivíduos da classe *TestTool*.

  $$\texttt{TestEnvironmentDevelopment} \sqsubseteq \forall\, \texttt{selectsTestTool}\, .\, \texttt{TestTool}$$

- ***TestLogProduction:*** esta atividade procura registrar como o teste foi conduzido. Anomalias que não são classificadas como incorretudes do *software*

podem ser também registradas durante a atividade de *test log production*.
Condição Necessária:

(i) *TestLogProduction* é subclasse de *TestActivity*.

$$\texttt{TestLogProduction} \sqsubseteq \texttt{TestActivity}$$

(ii) *TestLogProduction* sucede a atividade de avaliação dos resultados do teste. Indivíduos da classe *TestLogProduction* só relacionam-se através da propriedade *succeedsActivity* com indivíduos da classe *TestResultEvaluation*.

$$\texttt{TestLogProduction} \sqsubseteq \forall\ \texttt{succeedsActivity}\ .\ \texttt{TestResultEvaluation}$$

- ***TestResultEvaluation:*** a avaliação dos resultados determina se o teste atingiu as metas ou não. Em alguns casos, o sucesso significa que o *software* foi executado como esperado e não produziu nenhuma saída inesperada. Entretanto, nem todas as saídas inesperadas representam necessariamente uma falha.
Condição Necessária e Suficiente:

(i) Se algo avalia pelo menos um caso de teste em relação a pelo menos um valor obtido após a execução do teste, trata-se de uma atividade de avaliação dos resultados de teste e vice-versa.

$$\texttt{TestResultEvaluation} \equiv \begin{pmatrix} \exists\ \texttt{evaluatesTestCase}(\texttt{TestCase} \sqcap \\ (\exists\ \texttt{hasValue}\ .\ \texttt{TestCaseObtainedValue})) \end{pmatrix}$$

Condição Necessária:

(i) *TestResultEvaluation* é subclasse de *TestActivity*.

$$\texttt{TestResultEvaluation} \sqsubseteq \texttt{TestActivity}$$

(ii) A atividade de avaliação do teste antecede a atividade de registro dos defeitos ou registro de ocorrências. Indivíduos da classe *TestResultEvaluation* só relacionam-se através da propriedade *antecedesActivity* com indivíduos da classe união entre *TestIncorrectnessTracking* e *TestLogProduction*.

$$\texttt{TestResultEvaluation} \sqsubseteq \begin{pmatrix} \forall\ \texttt{antecedesActivity}\ .\ (\texttt{TestIncorrectnessTracking} \\ \sqcup\ \texttt{TestLogProduction}) \end{pmatrix}$$

(iii) A atividade de avaliação do teste sucede as atividades de planejamento ou execução. Indivíduos da classe *TestResultEvaluation* só relacionam-se com indivíduos da classe união entre *TestPlanning* e *TestExecution*.

$$\texttt{TestResultEvaluation} \sqsubseteq \forall\ \texttt{succeedsActivity}\ .\ (\texttt{TestPlanning} \sqcup \texttt{TestExecution})$$

(iv) Indivíduo da classe *TestResultEvaluation* relacionam-se através da propriedade *succedActivity* com pelo menos um indivíduo da classe *TestPlanning*.

$$\texttt{TestResultEvaluation} \sqsubseteq \exists\ \texttt{succeedsActivity}\ .\ \texttt{TestPlanning}$$

(v) Indivíduos da classe *TestResultEvaluation* relacionam-se através da propriedade *succeedActivity* com pelo menos um indivíduo da classe *TestExecution*.

$$\texttt{TestResultEvaluation} \sqsubseteq \exists\ \texttt{succeedsActivity}\ .\ \texttt{TestExecution}$$

As condições descritas nos itens (iii), (iv) e (v) constituem um *closure axiom*.

- ***TestDocumentation:*** Esta classe representa toda e qualquer documentação gerada no processo de teste. A documentação de teste pode ser produzida e continuamente mantida.
Condição Necessária:

  (i) *TestDocumentation* é subclasse de *SoftwareTestDomainConcept*.

$$\texttt{TestDocumentation} \sqsubseteq \texttt{SoftwareTestDomainConcept}$$

  (ii) Indivíduos da classe *TestDocumentation* só relacionam-se através da propriedade *isTestDocumentationOf* com indivíduos da classe *SoftwareTestProcess*.

$$\texttt{TestDocumentation} \sqsubseteq \forall\ \texttt{isTestDocumentationOf}\ .\ \texttt{SoftwareTestProcess}$$

  (iii) Indivíduos da classe *TestDocumentation* relacionam-se através da propriedade *isTestDocumentationOf* com pelo menos um indivíduo da classe *SoftwareTestProcess*.

$$\texttt{TestDocumentation} \sqsubseteq \exists\ \texttt{isTestDocumentationOf}\ .\ \texttt{SoftwareTestProcess}$$

  Disjunções - *Sobre as classes disjuntas de TestDocumentation, pode-se dizer:*

  (i) A classe *TestDocumentation* é disjunta das classes, *TestActivity*, *SoftwareTestProcess*, *TestLevel* e *TestPattern*, já que não compartilha instâncias com as mesmas.

$$\texttt{TestDocumentation} \sqsubseteq \begin{array}{c} \neg\ \texttt{TestActivity} \sqcap \\ \neg\ \texttt{SoftwareTestProcess} \sqcap \\ \neg\ \texttt{TestLeve} \sqcap \\ \neg\ \texttt{TestPattern} \end{array}$$

A Figura 5.9 introduz as subclasses de *TestDocumentation* que serão detalhadas a seguir. Todas elas são disjuntas entre si.

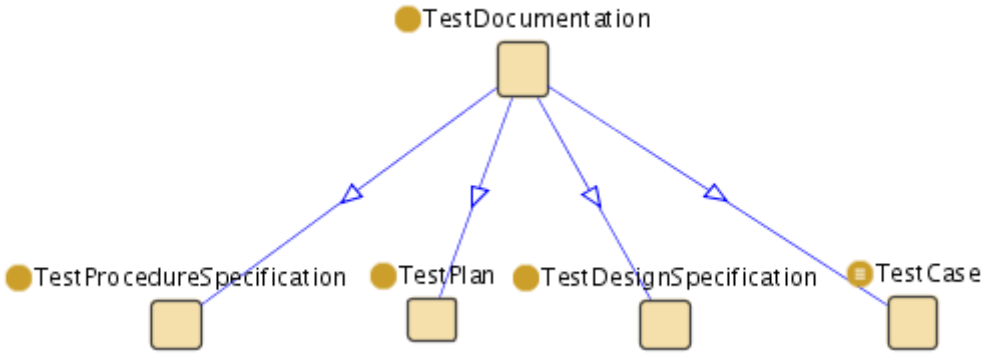

Figura 5.9: *TestDocumentation* e suas subclasses

- ***TestCase:*** esta classe representa o conjunto dos casos de teste. Um caso de teste representa uma possível forma de exercitar uma parte específica do

software. Um caso de teste concentra as especificações para o início e conclusão do teste.
Condição Necessária e Suficiente:

(i) Algo é inferido como um indivíduo da classe *TestCase* se e somente se relacionar-se através da propriedade *isGeneratedBy* com indivíduos da classe *TestCaseGeneration* e com pelo menos um indivíduo desta mesma classe através da propriedade *isGeneratedBy*.

$$\texttt{TestCase} \equiv \forall\ \texttt{isGeneratedBy}\ .\ \texttt{TestCaseGeneration}$$
$$\texttt{TestCase} \equiv \geq\ 1\ \texttt{isGeneratedBy}\ .\ \texttt{TestCaseGeneration}$$

Condição Necessária:

(i) *TestCase* é subclasse de *TestDocumentation*.

$$\texttt{TestCase} \sqsubseteq \texttt{TestDocumentation}$$

(ii) Indivíduos da classe *TestCase* só relacionam-se através da propriedade *hasProcedure* com indivíduos da classe *TestCaseProcedure* e com pelo menos um indivíduo desta classe através da mesma propriedade.

$$\texttt{TestCase} \sqsubseteq \forall\ \texttt{hasProcedure}\ .\ \texttt{TestCaseProcedure}$$
$$\texttt{TestCase} \sqsubseteq \exists\ \texttt{hasProcedure}\ .\ \texttt{TestCaseProcedure}$$

(iii) Indivíduos da classe *TestCase* só relacionam-se através da propriedade *canBeMappedInto* com indivíduos da classe *OSOnto:UseCase* e com pelo menos um indivíduo desta classe através da mesma propriedade.

$$\texttt{TestCase} \sqsubseteq \forall\ \texttt{canBeMappedInTo}\ .\ \textit{OSOnto:UseCase}$$
$$\texttt{TestCase} \sqsubseteq \exists\ \texttt{canBeMappedInTo}\ .\ \textit{OSOnto:UseCase}$$

(iv) O caso de teste só é detalhado após a especificação do projeto do teste. Indivíduos da classe *TestCase* só relacionam-se através da propriedade *succeedsDocumentation* com indivíduos da classe *TestDesignSpecification*.

$$\texttt{TestCase} \sqsubseteq \forall\ \texttt{succeedsDocumentation}\ .\ \texttt{TestDesignSpecification}$$

Propriedades - *Sobre as propriedades que descrevem a classe TestCase, pode-se dizer:*

(i) *antecedesDocumentation* é uma propriedade objeto inversa de *succeedsDocumentation* e transitiva.

$$\texttt{antecedesDocumentation} \in \texttt{P}_0$$
$$\texttt{antecedesDocumentation} \equiv \texttt{succeedsDocumentation}^-$$
$$\texttt{antecedesDocumentation} \in R_+$$

(ii) *isGeneratedBy* é uma propriedade objeto inversa de *generatesTestCase*, cujo domínio

é *TestCase* e o contradomínio é *TestCaseGeneration*.

$$\begin{aligned} \texttt{isGeneratedBy} &\in \texttt{P}_\texttt{O} \\ \texttt{isGeneratedBy} &\equiv \texttt{generatesTestCase}^- \\ \top &\sqsubseteq \begin{pmatrix} \forall\, \texttt{isGeneratedBy} \,.\, \texttt{TestCase} \\ (\texttt{TestCase} \neq \bot) \end{pmatrix} \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{isGeneratedBy}^- \,. \\ \texttt{TestCaseGeneration} \end{pmatrix} \end{aligned}$$

(iii) *hasProcedure* é uma subpropriedade objeto de *hasPart* inversa de *isProcedureOf*, cujo domínio é *TestCase* e o contradomínio é *TestCaseProcedure*.

$$\begin{aligned} \texttt{hasProcedure} &\in \texttt{P}_\texttt{O} \\ \texttt{hasProcedure} &\sqsubseteq \texttt{hasPart} \\ \texttt{hasProcedure} &\equiv \texttt{isProcedureOf}^- \\ \top &\sqsubseteq \forall\, \texttt{hasProcedure} \,.\, \texttt{TestCase}\ (\texttt{TestCase} \neq \bot) \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasProcedure}^- \,.\, \texttt{TestCaseProcedure} \\ (\texttt{TestCaseProcedure} \neq \bot) \end{pmatrix} \end{aligned}$$

(iv) *succeedsDocumentation* é uma propriedade objeto inversa de *antecedesDocumentation* e transitiva.

$$\begin{aligned} \texttt{succeedsDocumentation} &\in \texttt{P}_\texttt{O} \\ \texttt{succeedsDocumentation} &\equiv \texttt{antecedesDocumentation}^- \\ \texttt{succeedsDocumentation} &\in R_+ \end{aligned}$$

(v) *hasInputSpecification* é uma propriedade de tipo de dados cujo domínio especificado é a classe *TestCase* e o contradomínio é *XMLSchema:string*. Esta propriedade representa a especificação de entrada para um caso de teste.

$$\begin{aligned} \texttt{hasInputSpecification} &\in \texttt{P}_\texttt{D} \\ \top &\sqsubseteq \forall\, \texttt{hasInputSpecification} \,.\, \texttt{TestCase}\ (\texttt{TestCase} \neq \bot) \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasInputSpecification}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \end{aligned}$$

(vi) *hasOutputSpecification* é uma propriedade de tipo de dados cujo domínio especificado é a classe *TestCase* e o contradomínio é *XMLSchema:string*. Esta propriedade representa a especificação de saída de um caso de teste.

$$\begin{aligned} \texttt{hasOutputSpecification} &\in \texttt{P}_\texttt{D} \\ \top &\sqsubseteq \forall\, \texttt{hasOutputSpecification} \,.\, \texttt{TestCase}\ (\texttt{TestCase} \neq \bot) \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasOutputSpecification}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \end{aligned}$$

(vii) *hasPurpose* é uma propriedade de tipo de dados cujo domínio especificado é a classe *TestCase* e o contradomínio é *XMLSchema:string*. Representa o propósito de um caso

de teste.

$$\begin{aligned} \texttt{hasPurpose} &\in \texttt{P}_\texttt{D} \\ \top &\sqsubseteq \forall\, \texttt{hasPurpose} \,.\, \texttt{TestCase}\ (\texttt{TestCase} \neq \bot) \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasPurpose}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \end{aligned}$$

Instanciação - *A seguir temos algumas instâncias da classe TestCase.*

(i) Como foi discutido no Capítulo 4, o domínio de aplicação desta pesquisa é o teste do Linux. Para tanto, nos concentramos no LTP como repositório de teste. Neste repositório é possível encontrar uma série de casos de teste para os objetos (*dentry, file, inode,* e *superblock*) do sistema de arquivo virtual do Linux. As instâncias apresentadas a seguir, para o objeto *file*, são casos de teste reais extraídos do LTP.

$$\left\{ \begin{array}{l} \texttt{ftest01} \\ \texttt{ftest02} \\ \texttt{ftest03} \\ \texttt{ftest04} \\ \texttt{ftest05} \\ \texttt{ftest06} \\ \texttt{ftest07} \\ \texttt{ftest08} \end{array} \right\} \sqsubseteq \mathit{TestCase}$$

Cada instância representada aqui equivale no LTP ao código-fonte de um caso de teste implementado na linguagem C. Boa parte da documentação dos casos de teste está dispersa no próprio código-fonte e isso demanda um certo tempo e habilidade do desenvolvedor na hora de consultá-la e utilizá-la. As instâncias podem ser entendidas como arquivos dentro de um diretório. Para um usuário que não tem um conhecimento prévio do que é o *ftest01*, por exemplo, seria necessário abrir o aquivo, e observar o código-fonte, bem como os comentários sucintos ao longo do mesmo para saber do que se trata.
Por intermédio da formalização oferecida pela SwTO^I *ftest01* pode ser inferido como um caso de teste, pode ser encontrado facilmente através de um *plug-in* de consulta para a linguagem OWL DL e compartilhado, não mais sujeito à interpretação de cada pessoa mas representando o consenso da comunidade. Com o auxílio desta ontologia,

o *ftest01* pode ser compreendido através da especificação:

$$\begin{gathered}
\texttt{isGeneratedBy}(\texttt{ftest01}, \texttt{IBM_TestCaseGenerationProcess}) \\
\texttt{isTestDocumentationOf}(\texttt{ftest01}, \texttt{LinuxTestProcess}) \\
\texttt{hasInputSpecification}(\texttt{ftest01}, \texttt{lseek}) \\
\texttt{hasInputSpecification}(\texttt{ftest01}, \texttt{read}) \\
\texttt{hasInputSpecification}(\texttt{ftest01}, \texttt{write}) \\
\texttt{hasInputSpecification}(\texttt{ftest01}, \texttt{truncate}) \\
\texttt{hasInputSpecification}(\texttt{ftest01}, \texttt{ftruncate}) \\
\texttt{hasInputSpecification}(\texttt{ftest01}, \texttt{fsync}) \\
\texttt{hasInputSpecification}(\texttt{ftest01}, \texttt{sync}) \\
\texttt{hasInputSpecification}(\texttt{ftest01}, \texttt{fstat}) \\
\texttt{hasOutputSpecification}(\texttt{ftest01}, \text{flag 0 para saída inesperada}) \\
\texttt{hasPurpose}(\texttt{ftest01}, \begin{pmatrix} \text{Testar operações de} \\ \text{Input Output sobre um arquivo} \end{pmatrix}) \\
\texttt{hasProcedure}(\texttt{ftest01}, \text{lê}) \\
\texttt{hasProcedure}(\texttt{ftest01}, \text{escreve}) \\
\texttt{hasProcedure}(\texttt{ftest01}, \text{apaga}) \\
\texttt{canBeMappedInto}(\texttt{ftest01}, \texttt{ReadFile}) \\
\texttt{canBeMappedInto}(\texttt{ftest01}, \texttt{WriteFile}) \\
\texttt{canBeMappedInto}(\texttt{ftest01}, \texttt{RemoveFile})
\end{gathered}$$

- ***TestDesignSpecification:*** esta classe representa a especificação do projeto de teste, a qual refina a abordagem do teste, aponta os requisitos que devem ser cobertos, identifica os casos de teste e o *oracle*[3] que será utilizado.
  Condição Necessária:

  (i) *TestDesignSpecification* é subclasse de *TestDocumentation*.

  $$\texttt{TestDesignSpecification} \sqsubseteq \texttt{TestDocumentation}$$

  (ii) A especificação do projeto de teste antecede a documentação de casos de teste. Desta forma, indivíduos da classe *TestDesignSpecification* só relacionam-se através da propriedade *antecedesDocumentation* com indivíduos da classe *TestCase*.

  $$\texttt{TestDesignSpecification} \sqsubseteq \forall\ \texttt{antecedesDocumentation}\ .\ \texttt{TestCase}$$

  (iii) A especificação do projeto de teste procura informar que critérios de seleção de teste serão utilizados. Indivíduos da classe *TestDesignSpecification* só relacionam-se através da propriedade *specifiesSelectionCriteria* com indivíduos da classe *TestCaseSelectionCriteria*.

  $$\texttt{TestDesignSpecification} \sqsubseteq \begin{pmatrix} \forall\ \texttt{specifiesSelectionCriteria}\ . \\ \texttt{TestCaseSelectionCriteria} \end{pmatrix}$$

  (iv) Pelo menos um critério de seleção de teste é informado pelo documento de projeto de teste. Indivíduos da classe *TestDesignSpecification* relacionam-se através da propriedade *specifiesSelectionCriteria* com pelo menos um indivíduo da classe *TestCaseSelectionCriteria*.

  $$\texttt{TestDesignSpecification} \sqsubseteq \begin{pmatrix} \exists\ \texttt{specifiesSelectionCriteria}\ . \\ \texttt{TestCaseSelectionCriteria} \end{pmatrix}$$

[3] O *oracle* é um agente humano ou um software que define se um determinado teste passou ou falhou. Conforme discutido na Seção 2.3, o LTP dispõe de um *oracle*.

(v) O projeto de teste especifica o agente que fará parte do teste. Desta forma, indivíduos da classe *TestDesignSpecification* só relacionam-se através da propriedade *specifiesAgent* com indivíduos da classe *Agent*.

$$\texttt{TestDesignSpecification} \sqsubseteq \forall\, \texttt{specifiesAgent} \,.\, \texttt{Agent}$$

(vi) O projeto de teste sucede o plano de teste. Desta forma, indivíduos da classe *TestDesignSpecification* só relacionam-se através da propriedade *succeedsDocumentation* com indivíduos da classe *TestPlan*.

$$\texttt{TestDesignSpecification} \sqsubseteq \forall\, \texttt{succeedsDocumentation} \,.\, \texttt{TestPlan}$$

(vii) O projeto de teste leva em consideração os requisitos de *software* e pelo menos uma especificação destes requisitos. Desta forma, indivíduos da classe *TestDesignSpecification* só relacionam-se através da propriedade *considersRequirement* com indivíduos da classe *OSOnto:SoftwareRequirement* e com pelo menos um indivíduo desta classe através da mesma propriedade.

$$\texttt{TestDesignSpecification} \sqsubseteq \forall\, \texttt{considersRequirement} \,.\, \textit{OSOnto:SoftwareRequirement}$$
$$\texttt{TestDesignSpecification} \sqsubseteq \exists\, \texttt{considersRequirement} \,.\, \textit{OSOnto:SoftwareRequirement}$$

Propriedades - *Sobre as propriedades que definem a classe TestDesignSpecification, pode-se dizer:*

(i) *specifiesSelectionCriteria* é uma propriedade objeto, cujo domínio é *TestDesignSpecification* e o contradomínio é *TestCaseSelectionCriteria*.

$$\texttt{specifiesSelectionCriteria} \in \texttt{P}_0$$
$$\top \sqsubseteq \forall\left(\begin{array}{c}\texttt{specifiesSelectionCriteria} \,.\, \texttt{TestDesignSpecification} \\ (\texttt{TestDesignSpecification} \neq \bot)\end{array}\right)$$
$$\top \sqsubseteq \forall\left(\begin{array}{c}\texttt{specifiesSelectionCriteria}^- \,.\, \texttt{TestCaseSelectionCriteria} \\ (\texttt{TestCaseSelectionCriteria} \neq \bot)\end{array}\right)$$

(ii) *specifiesAgent* é uma propriedade objeto, cujo domínio é *TestDesignSpecification* e o contradomínio é *Agent*.

$$\texttt{specifiesAgent} \in \texttt{P}_0$$
$$\top \sqsubseteq \forall\left(\begin{array}{c}\texttt{specifiesAgent} \,.\, \texttt{TestDesignSpecification} \\ (\texttt{TestDesignSpecification} \neq \bot)\end{array}\right)$$
$$\top \sqsubseteq \forall\, \texttt{specifiesAgent}^- \,.\, \texttt{Agent}\ (\texttt{Agent} \neq \bot)$$

- ***TestPlan:*** representa o conjunto dos planos de teste. Um plano de teste descreve o escopo e abordagem do teste, recursos que serão utilizados, as atividades que serão realizadas, as pessoas ou softwares responsáveis por cada atividade.
Condição Necessária:

(i) *TestPlan* é subclasse de *TestDocumentation*.

$$\texttt{TestPlan} \sqsubseteq \texttt{TestDocumentation}$$

(ii) O plano de teste antecede o projeto de teste já que é o plano que dá as diretrizes do projeto. Desta forma, indivíduos da classe *TestPlan* só relacionam-se através da pro-

priedade *antecedesDocumentation* com indivíduos da classe *TestDesignSpecification*.

$$\texttt{TestPlan} \sqsubseteq \forall\ \texttt{antecedesDocumentation}\ .\ \texttt{TestDesignSpecification}$$

(iii) O plano de teste é criado por pelo menos uma atividade de planejamento. Desta forma, indivíduos da classe *TestPlan* relacionam-se através da propriedade *isCreatedBy* com pelo menos um indivíduo da classe *TestPlanning*.

$$\texttt{TestPlan} \sqsubseteq \exists\ \texttt{isCreatedBy}\ .\ \texttt{TestPlanning}$$

(iv) O plano de teste procura estabelecer os critérios selecionados para o processo de teste. Desta forma, indivíduos da classe *TestPlan* relacionam-se através da propriedade *organizesTestCriteria* com indivíduos da classe união entre *TestGuide, TestLevel, TestPattern* e *TestTechnique*.

$$\texttt{TestPlan} \sqsubseteq \exists\ \texttt{organizesTestCriteria} \begin{pmatrix} \texttt{TestGuide} \sqcup \\ \texttt{TestLevel} \sqcup \\ \texttt{TestPattern} \sqcup \\ \texttt{TestTechnique} \end{pmatrix}$$

(v) O plano de teste procura organizar as atividades que serão realizadas no processo de teste que por sua vez, são atividades que podem ser realizadas por pessoas ou por um *software*. Desta forma, a classe *TestPlan* é subclasse da classe anônima que comporta indivíduos que se relacionam através da propriedade *organizesActivity* com pelo menos um indivíduo da classe *TestActivity* que por sua vez, são indivíduos que se relacionam através da propriedade *isPerformedBy* com pelo menos um indivíduo da classe união entre *TestTeam* e *OSOnto:Software*. A Figura 5.10 ilustra esta definição da classe *TestPlan*.

$$\texttt{TestPlan} \sqsubseteq \begin{pmatrix} \exists\ \texttt{organizesActivity}(\texttt{TestActivity} \sqcap \\ (\exists\ \texttt{isPerformedBy}(\texttt{TestTeam} \sqcup \texttt{OSOnto} : \texttt{Software}))) \end{pmatrix}$$

Propriedades - *Sobre as propriedades que descrevem a classe TestPlan, pode-se dizer:*

(i) *isCreatedBy* é uma propriedade objeto inversa de *createsTestPlan*.

$$\begin{aligned} \texttt{isCreatedBy} &\in \texttt{P}_\texttt{O} \\ \texttt{isCreatedBy} &\equiv \texttt{createsTestPlan}^- \end{aligned}$$

(ii) *hasTestApproach* é uma propriedade de tipo de dados cujo domínio é a classe *TestPlan* e o contradomínio é *XMLSchema:string*. Esta propriedade representa uma possível abordagem adotada para o teste descrito pelo plano.

$$\begin{aligned} \texttt{hasTestApproach} &\in \texttt{P}_\texttt{D} \\ \top &\sqsubseteq \forall\ \texttt{hasTestApproach}\ .\ \texttt{TestPlan}\ (\texttt{TestPlan} \neq \bot) \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasTestApproach}^-\ .\ \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \end{aligned}$$

(iii) *hasTestFeature* é uma propriedade de tipo de dados cujo domínio especificado é a classe *TestPlan* e o contradomínio é *XMLSchema:string*. Esta propriedade representa

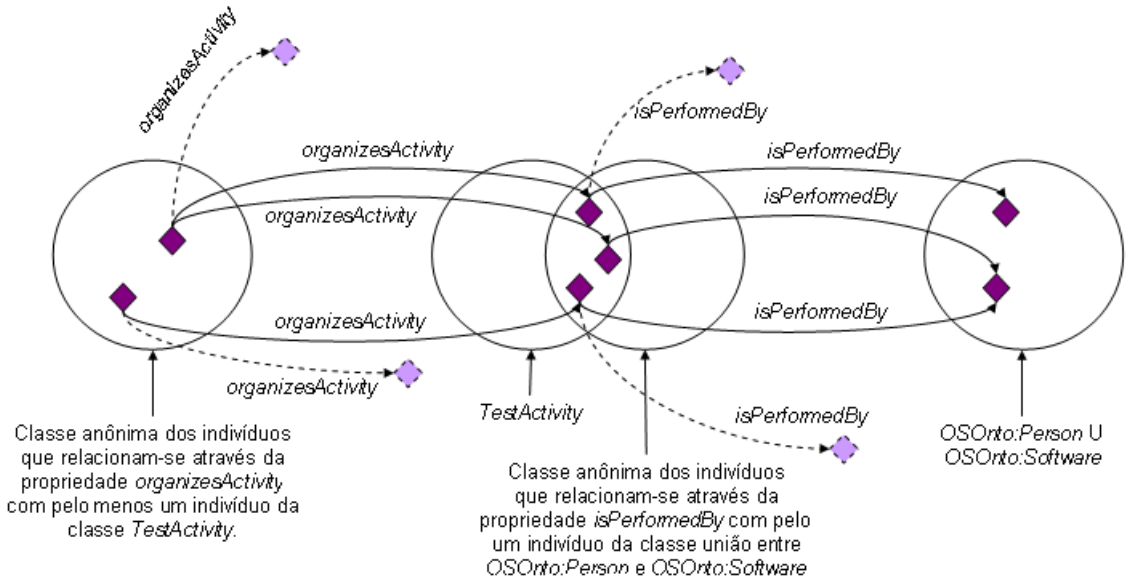

Figura 5.10: Uma descrição da classe *TestPlan*

as características peculiares do teste que são descritas no plano.

$$\begin{aligned} \texttt{hasTestFeature} &\in \texttt{P}_\texttt{D} \\ \top &\sqsubseteq \forall\, \texttt{hasTestFeature} \,.\, \texttt{TestPlan}\ (\texttt{TestPlan} \neq \bot) \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasTestFeature}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \end{aligned}$$

(iv) *hasTestResource* é uma propriedade de tipo de dados cujo domínio especificado é a classe *TestPlan* e o contradomínio é *XMLSchema:string*. Esta propriedade representa os recursos do teste que são descritos pelo plano.

$$\begin{aligned} \texttt{hasTestResource} &\in \texttt{P}_\texttt{D} \\ \top &\sqsubseteq \forall\, \texttt{hasTestResource} \,.\, \texttt{TestPlan}\ (\texttt{TestPlan} \neq \bot) \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasTestResource}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \end{aligned}$$

(v) *hasTestRisk* é uma propriedade de tipo de dados cujo domínio especificado é a classe *TestPlan* e o contradomínio é *XMLSchema:string*. Esta propriedade representa possíveis riscos associados ao teste que são elicitados pelo plano.

$$\begin{aligned} \texttt{hasTestRisk} &\in \texttt{P}_\texttt{D} \\ \top &\sqsubseteq \forall\, \texttt{hasTestRisk} \,.\, \texttt{TestPlan}\ (\texttt{TestPlan} \neq \bot) \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasTestRisk}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \end{aligned}$$

(vi) *hasTestScope* é uma propriedade de tipo de dados cujo domínio especificado é a classe *TestPlan* e o contradomínio é *XMLSchema:string*. Esta propriedade representa o

escopo do teste.

$$\begin{aligned} \texttt{hasTestScope} &\in \texttt{P}_\texttt{D} \\ \top &\sqsubseteq \forall\, \texttt{hasTestScope} \,.\, \texttt{TestPlan}\ (\texttt{TestPlan} \neq \bot) \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasTestScope}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \end{aligned}$$

- ***TestCaseProcedure:*** representa o conjunto de passos requeridos para operar o sistema e exercitar os casos de teste especificados no projeto. Cada procedimento do caso de teste pode requerer um conjunto de variáveis de teste.
  Condição Necessária:

  (i) *TestCaseProcedure* é subclasse de *TestDocumentation*.

  $$\texttt{TestCaseProcedure} \sqsubseteq \texttt{TestDocumentation}$$

  (ii) A especificação do procedimento de teste representa o conjunto de passos de um caso de teste. Indivíduos da classe *TestCaseProcedure* só relacionam-se através da propriedade *isProcedureOf* com indivíduos da classe *TestCase*.

  $$\texttt{TestCaseProcedure} \sqsubseteq \forall\, \texttt{isProcedureOf} \,.\, \texttt{TestCase}$$

  (iii) Indivíduos da classe *TestCaseProcedure* relacionam-se através da propriedade *isProcedureOf* com pelo menos um indivíduo da classe *TestCase*.

  $$\texttt{TestCaseProcedure} \sqsubseteq \exists\, \texttt{isProcedureOf} \,.\, \texttt{TestCase}$$

  (iv) As especificações do procedimento de teste relacionam-se com variáveis de teste. Desta forma, indivíduos da classe *TestCaseProcedure* só relacionam-se através da propriedade *containsTestVariable* com indivíduos da classe *TestVariable*.

  $$\texttt{TestCaseProcedure} \sqsubseteq \forall\, \texttt{containsTestVariable} \,.\, \texttt{TestVariable}$$

  (v) Uma especificação do procedimento de teste sucede a documentação dos casos de teste. Desta forma, indivíduos da classe *TestCaseProcedure* só relacionam-se através da propriedade *succeedsDocumentation* com indivíduos da classe *TestCase*.

  $$\texttt{TestCaseProcedure} \sqsubseteq \forall\, \texttt{succeedsDocumentation} \,.\, \texttt{TestCase}$$

  Propriedades - *Sobre as propriedades que descrevem a classe TestCaseProcedure, pode-se dizer:*

  (i) *isProcedureOf* é uma subpropriedade objeto de *isPartOf* inversa de *hasProcedure*, cujo domínio é *TestCaseProcedure* e o contradomínio é *TestCase*.

  $$\begin{aligned} \texttt{isProcedureOf} &\in \texttt{P}_\texttt{O} \\ \texttt{isProcedureOf} \sqsubseteq \texttt{isPartOfisProcedureOf} &\equiv \texttt{hasProcedure}^- \\ \top &\sqsubseteq \forall\, \texttt{isProcedureOf} \,.\, \texttt{TestCaseProcedure} \\ \top &\sqsubseteq \forall\, \texttt{isProcedureOf}^- \,.\, \texttt{TestCase} \end{aligned}$$

  (ii) *containsTestVariable* é uma propriedade objeto inversa de *isTestVariableOf*, cujo domínio

é *TestCaseProcedure* e o contradomínio é *TestVariable*.

$$\begin{aligned} \texttt{containsTestVariable} &\in \texttt{P}_0 \\ \texttt{containsTestVariable} &\equiv \texttt{isTestVariableOf}^- \\ \top &\sqsubseteq \forall\, \texttt{containsTestVariable} \,.\, \texttt{TestCaseProcedure} \\ \top &\sqsubseteq \forall\, \texttt{containsTestVariable}^- \,.\, \texttt{TestVariable} \end{aligned}$$

- ***TestFundamental:*** classe que representa fundametos de teste de *software*.
  Condições Necessárias:

  (i) *TestFundamental* é subclasse de *SoftwareTestDomainConcept*.

  $$\texttt{TestFundamental} \sqsubseteq \texttt{SoftwareTestDomainConcept}$$

  (ii) Indivíduos da classe *TestFundamental* só relacionam-se através da propriedade *isTestingFundamentalOf* com indivíduos da classe *SoftwareTestProcess*. E pela mesma propriedade, relacionam-se com pelo menos um indivíduo da classe *SoftwareTestProcess*.

  $$\begin{aligned} \texttt{TestFundamental} &\sqsubseteq \forall\, \texttt{isTestFundamentalOf} \,.\, \texttt{SoftwareTestProcess} \\ \texttt{TestFundamental} &\sqsubseteq \exists\, \texttt{isTestFundamentalOf} \,.\, \texttt{SoftwareTestProcess} \end{aligned}$$

  Disjunções - *Sobre as classes disjuntas de TestFundamental, pode-se dizer:*

  (i) A classe *TestFundamental* é disjunta das classes, *TestActivity*, *SoftwareTestProcess*, *TestMeasurementMethod*, *TestLevel* e *TestTechnique* já que não compartilha instâncias com as mesmas.

  $$\begin{aligned} \texttt{TestFundamental} \sqsubseteq\ & \neg\, \texttt{TestActivity} \sqcap \\ & \neg\, \texttt{SoftwareTestProcess} \sqcap \\ & \neg\, \texttt{TestMeasurementMethod} \sqcap \\ & \neg\, \texttt{TestLevel} \sqcap \\ & \neg\, \texttt{TestTechnique} \end{aligned}$$

  A Figura 5.11 introduz as subclasses de *TestFundamental* que serão detalhadas a seguir.

- ***Agent:*** esta classe representa o conjunto de agentes que atuam em um processo de teste de *software*.
  Condições Necessárias:

  (i) *Agent* é subclasse de *TestFundamental*.

  $$\texttt{Agent} \sqsubseteq \texttt{TestFundamental}$$

  (ii) Indivíduos da classe *Agent* só relacionam-se através da propriedade *interactsWithAgent* com indivíduos da classe *Agent*.

  $$\texttt{Agent} \sqsubseteq \forall\, \texttt{interactsWithAgent} \,.\, \texttt{Agent}$$

  (iii) Um agente pode ser uma pessoa ou um software.

  $$\texttt{Agent} \sqsubseteq \texttt{OSOnto} : \texttt{Person} \sqcup \texttt{OSOnto} : \texttt{Software}$$

  Disjunções - *Sobre as classes disjuntas de Agent, pode-se dizer:*

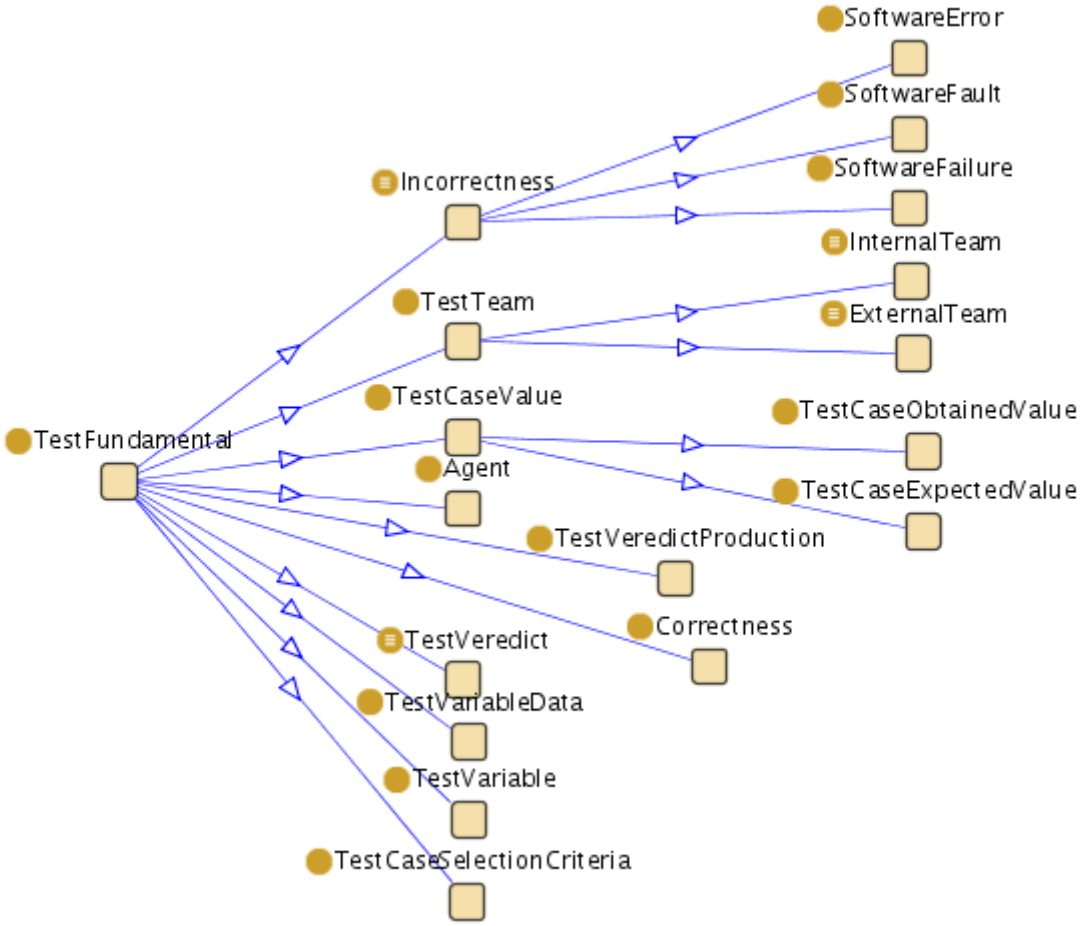

Figura 5.11: *TestFundamental* e suas subclasses

(i) A classe *Agent* é disjunta das classes *TestVariableData*, *TestVariable* e *TestVeredict*, já que não compartilha instâncias com as mesmas.

$$\texttt{Agent} \sqsubseteq \begin{array}{l} \neg\ \texttt{TestVariableData} \sqcap \\ \neg\ \texttt{TestVariable} \sqcap \\ \neg\ \texttt{TestVeredict} \end{array}$$

- ***Correctness:*** esta classe representa o conceito de corretude.
  Condição Necessária:

  (i) *Correctness* é subclasse de *TestFundamental*.

  $$\texttt{Correctness} \sqsubseteq \texttt{TestFundamental}$$

  (ii) *Correctness* é subclasse do complemento de *Incorrectness*.

  $$\texttt{Correctness} \sqsubseteq \neg\ \texttt{Incorrectness}$$

Disjunção - *A classe Correctness é disjunta de Incorrectness já que estas classes não compartilham instâncias*

$$\texttt{Correctness} \sqsubseteq \neg\, \texttt{Incorrectness}$$

- ***Incorrectness:*** esta classe representa o conceito de incorretude.
Condição Necessária e Suficiente:

  (i) Indivíduo é uma incorretude se e somente se for um erro, um defeito ou uma falha.

  $$\texttt{Incorrectness} \equiv \texttt{SoftwareError} \sqcup \texttt{SoftwareDefect} \sqcup \texttt{SoftwareFault}$$

  A classe *Incorrectness* possui três subclasses disjuntas entre si:

  - *SoftwareError:* esta classe representa o conjunto dos erros ou discrepâncias entre o que foi computado, observado ou mensurado e o que é dito correto, ou seja, um erro pode ser facilmente identificado quando temos um parâmetro de corretude bem definido;
  - *SoftwareDefect:* esta classe representa o conjunto dos defeitos ou incapacidades do sistema para realizar uma função requerida de acordo com o requisito de performance, por exemplo;
  - *SoftwareFault:* classe que representa o conjunto das falhas. Uma falha geralmente causa um defeito. O teste freqüentemente identifica defeitos mas efetivamente são as falhas que devem ser removidas.

  A forma como esta condição foi definida também é conhecida como axioma de cobertura (em inglês, *covering axiom*). Por um axioma de cobertura é possível especificar que todos os indivíduos da classe *Incorrectness* também são indivíduos de uma das suas subclasses já que estas são disjuntas entre si.

  Condição Necessária:

  (i) *Incorrectness* é subclasse de *TestFundamental*.

  $$\texttt{Incorrectness} \sqsubseteq \texttt{TestFundamental}$$

  Disjunção - *A classe Incorrectness é disjunta de Correctness já que estas classes não compartilham instâncias*

  $$\texttt{Incorrectness} \sqsubseteq \neg\, \texttt{Correctness}$$

- ***TestVeredictProduction:*** esta classe representa a produção de um veredito de teste realizada por um agente sobre a execução de um *software*. Esta classe representa uma relação ternária entre *OSOnto:Software*, *Agent* e *TestVeredict*. Como em OWL as propriedades são binárias, para concretizar esta representação utilizamos a técnica de reificação (em inglês, *reified relationships*) que consiste, neste caso, em representar a relação ternária como uma classe conforme demonstrado a seguir.
Condição Necessária:

  (i) *TestVeredictProduction* é subclasse de *TestFundamental*.

  $$\texttt{TestVeredictProduction} \sqsubseteq \texttt{TestFundamental}$$

  (ii) Indivíduos da classe *TestVeredictProduction* só relacionam-se através da propriedade *byAgent* com indivíduos da classe *Agent* e com pelo menos um indivíduo desta classe

através da mesma propriedade.

$$\texttt{TestVeredictProduction} \sqsubseteq \forall\, \texttt{byAgent} \,.\, \texttt{Agent}$$
$$\texttt{TestVeredictProduction} \sqsubseteq \exists\, \texttt{byAgent} \,.\, \texttt{Agent}$$

(iii) Indivíduos da classe *TestVeredictProduction* só relacionam-se através da propriedade *hasVeredict* com indivíduos da classe *TestVeredict* e com pelo menos um indivíduo desta classe através da mesma propriedade.

$$\texttt{TestVeredictProduction} \sqsubseteq \forall\, \texttt{hasVeredict} \,.\, \texttt{TestVeredict}$$
$$\texttt{TestVeredictProduction} \sqsubseteq \exists\, \texttt{hasVeredict} \,.\, \texttt{TestVeredict}$$

(iv) Indivíduos da classe *TestVeredictProduction* só relacionam-se através da propriedade *forSoftware* com indivíduos da classe *OSOnto:Software* e com pelo menos um indivíduo desta classe através da mesma propriedade.

$$\texttt{TestVeredictProduction} \sqsubseteq \forall\, \texttt{forSoftware} \,.\, \texttt{OSOnto} : \texttt{Software}$$
$$\texttt{TestVeredictProduction} \sqsubseteq \exists\, \texttt{forSoftware} \,.\, \texttt{OSOnto} : \texttt{Software}$$

Propriedades - *Sobre as propriedades que definem a classe TestVeredictProduction, pode-se dizer:*

(i) *byAgent* é uma propriedade objeto, cujo domínio é *TestVeredictProduction* e o contradomínio é *Agent*.

$$\texttt{byAgent} \in \texttt{P}_\texttt{0}$$
$$\top \sqsubseteq \forall\, \texttt{byAgent} \,.\, \texttt{TestVeredictProduction}\ (\texttt{TestVeredictProduction} \neq \bot)$$
$$\top \sqsubseteq \forall\, \texttt{byAgent}^- \,.\, \texttt{Agent}\ (\texttt{Agent} \neq \bot)$$

(ii) *forSoftwareInTest* é uma propriedade objeto funcional, cujo domínio é *TestVeredictProduction* e o contradomínio é *OSOnto:Software*.

$$\texttt{forSoftwareInTest} \in \texttt{P}_\texttt{0}$$
$$\top \sqsubseteq (\leq\ 1\ \texttt{forSoftwareInTest})$$
$$\top \sqsubseteq \forall\left(\begin{array}{c}\texttt{forSoftwareInTest} \,.\, \texttt{TestVeredictProduction} \\ (\texttt{TestVeredictProduction} \neq \bot)\end{array}\right)$$
$$\top \sqsubseteq \forall\left(\begin{array}{c}\texttt{forSoftwareInTest}^- \,.\, \texttt{OSOnto} : \texttt{Software} \\ (\texttt{OSOnto} : \texttt{Software} \neq \bot)\end{array}\right)$$

(iii) *hasVeredict* é uma propriedade objeto funcional, cujo domínio é *TestVeredictProduction* e o contradomínio é *TestVeredict*.

$$\texttt{hasVeredict} \in \texttt{P}_\texttt{0}$$
$$\top \sqsubseteq (\leq\ 1\ \texttt{hasVeredict})$$
$$\top \sqsubseteq \forall\left(\begin{array}{c}\texttt{hasVeredict} \,.\, \texttt{TestVeredictProduction} \\ (\texttt{TestVeredictProduction} \neq \bot)\end{array}\right)$$
$$\top \sqsubseteq \forall\left(\begin{array}{c}\texttt{hasVeredict}^- \,.\, \texttt{TestVeredict} \\ (\texttt{TestVeredict} \neq \bot)\end{array}\right)$$

Disjunção - *A classe TestVeredictProduction é disjunta de TestVariableData, TestCaseSe-*

lectionCriteria, TestTeam e TestVariable já que não compartilha instâncias com as mesmas.

$$\texttt{TestVeredictProduction} \sqsubseteq \begin{array}{c} \neg\ \texttt{TestVariableData} \\ \neg\ \texttt{TestCaseSelectionCriteria} \\ \neg\ \texttt{TestTeam} \\ \neg\ \texttt{TestVariable} \end{array}$$

- ***TestVariableData:*** esta classe representa os possíveis dados de uma variável de teste.
  Condição Necessária:

  (i) *TestVariableData* é subclasse de *TestFundamental*.

  $$\texttt{TestVariableData} \sqsubseteq \texttt{TestFundamental}$$

  (ii) Indivíduos da classe *TestVariableData* só relacionam-se através da propriedade *isDataOf* com indivíduos da classe *TestVariable* e pela mesma propriedade, relacionam-se com pelo menos um indivíduo da classe *TestVariable*.

  $$\texttt{TestData} \sqsubseteq \forall\ \texttt{isDataOf}\ .\ \texttt{TestVariable}$$
  $$\texttt{TestVariableData} \sqsubseteq \exists\ \texttt{isDataOf}\ .\ \texttt{TestVariable}$$

- ***TestCaseSelectionCriteria:*** esta classe representa o conjunto dos critérios de seleção de casos de teste. O registro adequado do conhecimento do especialista para definição destes critérios pode auxiliar na decisão de que casos são mais adequados em certas circunstâncias.
  Condição Necessária:

  (i) *TestCaseSelectionCriteria* é subclasse de *TestFundamental*.

  $$\texttt{TestCaseSelectionCriteria} \sqsubseteq \texttt{TestFundamental}$$

  (ii) O critério de seleção de casos de teste é definido com base na estatégia de teste e nos tipos de teste de *software* conforme axioma de fechamento a seguir.

  $$\texttt{TestCaseSelectionCriteria} \sqsubseteq \begin{pmatrix} \forall\ \texttt{isBasedIn}\ (\texttt{TestStrategyBaseInSwDevelopment} \sqcup \\ \texttt{SoftwareTest}) \end{pmatrix}$$
  $$\texttt{TestCaseSelectionCriteria} \sqsubseteq \exists\ \texttt{isBasedIn}\ .\ \texttt{TestStrategyBaseInSwDevelopment}$$
  $$\texttt{TestCaseSelectionCriteria} \sqsubseteq \exists\ \texttt{isBasedIn}\ .\ \texttt{SoftwareTest}$$

  (iii) Indivíduos da classe *TestCaseSelectionCriteria* relacionam-se através da propriedade *selectsTestCase* com no mínimo um indivíduo da classe *TestCase*.

  $$\texttt{TestCaseSelectionCriteria} \sqsubseteq \geq\ 1\ \texttt{selectsTestCase}\ .\ \texttt{TestCase}$$

  Disjunções: a classe *TestCaseSelectionCriteria* é disjunta das classes *TestVariable* e *TestTeam* já que não compartilha instâncias com as mesmas.

  $$\texttt{TestCaseSelectionCriteria} \sqsubseteq \begin{array}{c} \neg\ \texttt{TestVariable} \sqcap \\ \neg\ \texttt{TestTeam} \end{array}$$

- ***TestTeam:*** esta classe representa o conjunto de equipes de teste formadas

por pessoas.
Condição Necessária e Suficiente:

(i) Um indivíduo é inferido como membro de uma equipe de teste se e somente se fizer parte de uma equipe externa ou interna conforme o axioma de cobertura a seguir.

$$\texttt{TestTeam} \equiv \texttt{ExternalTeam} \sqcup \texttt{InternalTeam}$$

ExternalTeam e *InternalTeam* são subclasses de *TestTeam*, disjuntas entre si:

- ***ExternalTeam:*** conjunto formado por pessoas externas ao projeto de desenvolvimento de *software* com uma pespectiva independente.
- ***InternalTeam:*** conjunto formado por pessoas que fazem parte do projeto de desenvolvimento de *software*, envolvidas ou não com sua implementação.

Condição Necessária:

(i) *TestTeam* é subclasse de *TestFundamental*.

$$\texttt{TestTeam} \sqsubseteq \texttt{TestFundamental}$$

(ii) Uma instância da classe *TestTeam* só relaciona-se através da propriedade *isFormedBy* com indivíduos da classe *OSOnto:Person* e com pelo menos um indivído desta classe através da mesma propriedade.

$$\texttt{TestTeam} \sqsubseteq \forall\, \texttt{isFormedBy} \,.\, \textbf{OSOnto:Person}$$
$$\texttt{TestTeam} \sqsubseteq \exists\, \texttt{isFormedBy} \,.\, \textbf{OSOnto:Person}$$

- ***TestVariable:*** representa o conjunto de variáveis usadas em cada procedimento de um caso de teste. Digamos que um dos procedimentos seja "entrar com uma senha", este passo pode dispor das seguintes variáveis de entrada: senha válida, inválida e nula. Para as variáveis *senha válida* e *senha inválida* é possível definir os dados de teste, definindo claramente o que é uma senha válida e o que é uma senha inválida.
Condição Necessária:

(i) *TestVariable* é subclasse de *TestFundamental*.

$$\texttt{TestVariable} \sqsubseteq \texttt{TestFundamental}$$

(ii) Indivíduos da classe *TestVariable* só relacionam-se através da propriedade *hasData* com indivíduos da classe *TestVariableData* e pela mesma propriedade, relacionam-se com pelo menos um indivíduo da classe *TestVariableData*.

$$\texttt{TestVariable} \sqsubseteq \forall\, \texttt{hasData} \,.\, \texttt{TestVariableData}$$
$$\texttt{TestVariable} \sqsubseteq \exists\, \texttt{hasData} \,.\, \texttt{TestVariableData}$$

(iii) Indivíduos da classe *TestVariable* só relacionam-se através da propriedade *isTestVariableOf* com indivíduos da classe *TestCaseProcedure* e pela mesma propriedade, relacionam-se com pelo menos um indivíduo da classe *TestCaseProcedure*.

$$\texttt{TestVariable} \sqsubseteq \forall\, \texttt{isTestVariableOf} \,.\, \texttt{TestCaseProcedure}$$
$$\texttt{TestVariable} \sqsubseteq \exists\, \texttt{isTestVariableOf} \,.\, \texttt{TestCaseProcedure}$$

- ***TestCaseValue:*** esta classe representa o conjunto de valores usados no domínio de casos de teste.
Condição Necessária:

  (i) *TestCaseValue* é subclasse de *TestFundamental*.

  $$\texttt{TestCaseValue} \sqsubseteq \texttt{TestFundamental}$$

  (ii) Um valor de um caso teste de um *software*, pode ser interpretado como correto ou incorreto. Desta forma, indivíduos desta classe só relacionam-se através da propriedade *isInterpretedAs* com indivíduos da classe resultante da união entre *Correctness* e *Incorrectness*.

  $$\texttt{TestCaseValue} \sqsubseteq \forall\ \texttt{isInterpretedAs}\ (\texttt{Correctness} \sqcup \texttt{Incorrectness})$$

  A classe *TestCaseValue* possui duas subclasses: *TestCaseExpectedValue* que representa o conjunto dos valores esperados para um caso de teste e *TestCaseObtainedValue* que representa o conjunto dos valores obtidos para um caso de teste após a sua execução. Estas classes não são disjuntas porque podem compartilhar instâncias, ou seja, um valor esperado pode ser classificado como obtido após uma execução.

- ***TestVeredict:*** esta classe representa os possíveis vereditos de um agente sobre a execução de um teste de *software*.
Condição Necessária e Suficiente:

  (i) Um veredito de um teste é equivalente a classe enumerada formada única e exclusivamente pelas instâncias *fail* e *pass*.

  $$\texttt{TestVeredict} \equiv \texttt{fail pass}$$

  Condição Necessária:

  (i) *TestVeredict* é subclasse de ***TestFundamental***.

  $$\texttt{TestVeredict} \sqsubseteq \texttt{TestFundamental}$$

- ***TestGuide:*** esta classe representa o conjunto dos guias de teste. As fases de teste podem ser guiadas por vários critérios. Por exemplo, os riscos identificados no projeto de um *software* podem auxiliar na definição de prioridades e focos das estratégias de teste.
Condição Necessária:

  (i) *TestGuide* é subclasse de *SoftwareTestDomainConcept*.

  $$\texttt{TestGuide} \sqsubseteq \texttt{SoftwareTestDomainConcept}$$

  (ii) Indivíduos da classe *TestGuide* só relacionam-se através da propriedade *isTestGuideOf* com indivíduos da classe *SoftwareTestProcess* e pela mesma propriedade, relacionam-se com pelo menos um indivíduo da classe *SoftwareTestProcess*.

  $$\texttt{TestGuide} \sqsubseteq \forall\ \texttt{isTestGuideOf}\ .\ \texttt{SoftwareTestProcess}$$
  $$\texttt{TestGuide} \sqsubseteq \exists\ \texttt{isTestGuideOf}\ .\ \texttt{SoftwareTestProcess}$$

Disjunção - *A classe TestGuide é disjunta de TestActivity, TestTechnique, SoftwareTestProcess e TestPattern já que estas classes não compartilham instâncias*

$$\texttt{TestGuide} \sqsubseteq \begin{array}{c} \neg\ \texttt{TestActivity} \\ \neg\ \texttt{TestTechnique} \\ \neg\ \texttt{SoftwareTestProcess} \\ \neg\ \texttt{TestPattern} \end{array}$$

- ***TestLevel:*** geralmente os testes são realizados em níveis diferentes ao longo do processo de desenvolvimento e manutenção do *software*. Nem todos os tipos e estratégias de teste se aplicam aos diversos tipos de *software*, por este motivo, o teste é particionado em níveis para facilitar a seleção apropriada do nível que será utilizado em cada processo de teste.
  Condição Necessária:

  (i) *TestLevel* é subclasse de *SoftwareTestDomainConcept*.

  $$\texttt{TestLevel} \sqsubseteq \texttt{SoftwareTestDomainConcept}$$

  (ii) Indivíduos da classe *TestLevel* só relacionam-se através da propriedade *isTestLevelOf* com indivíduos da classe *SoftwareTestProcess* e pela mesma propriedade, relacionam-se com pelo menos um indivíduo da classe *SoftwareTestProcess*.

  $$\texttt{TestLevel} \sqsubseteq \forall\ \texttt{isTestLevelOf}\ .\ \texttt{SoftwareTestProcess}$$
  $$\texttt{TestLevel} \sqsubseteq \exists\ \texttt{isTestLevelOf}\ .\ \texttt{SoftwareTestProcess}$$

  Disjunção - *A classe TestLevel é disjunta de TestActivity, TestDocumentation TestFundamental, SoftwareTestProcess e TestMeasurementMethod já que estas classes não compartilham instâncias*

  $$\texttt{TestLevel} \sqsubseteq \left(\begin{array}{c} \neg\ \texttt{TestActivity} \\ \neg\ \texttt{TestDocumentation} \\ \neg\ \texttt{TestFundamental} \\ \neg\ \texttt{SoftwareTestProcess} \\ \neg\ \texttt{TestMeasurementMethod} \end{array} \right)$$

  A Figura 5.12 introduz as subclasses de *TestLevel* que são disjuntas entre si.

- ***TestStrategyBaseInSwDevelopment:*** esta classe representa um nível de teste particionados segundo uma estratégia que leva em consideração o ciclo tradicional de desenvolvimento do software. Três grandes estratégias conceituais são freqüentemente referenciadas: teste unitário, teste de integração e teste de sistema.
  Condição Necessária:

  (i) *TestStrategyBaseInSwDevelopment* é subclasse de *TestGuide*.

  $$\texttt{TestStrategyBaseInSwDevelopment} \sqsubseteq \texttt{TestLevel}$$

  Disjunção - *A classe TestStrategyBaseInSwDevelopment é disjunta de SoftwareTest já que estas classes não compartilham instâncias*

  $$\texttt{TestStrategyBaseInSwDevelopment} \sqsubseteq \neg\ \texttt{SoftwareTest}$$

  A classe *TestStrategyBaseInSwDevelopment* possui três subclasses disjuntas entre si:

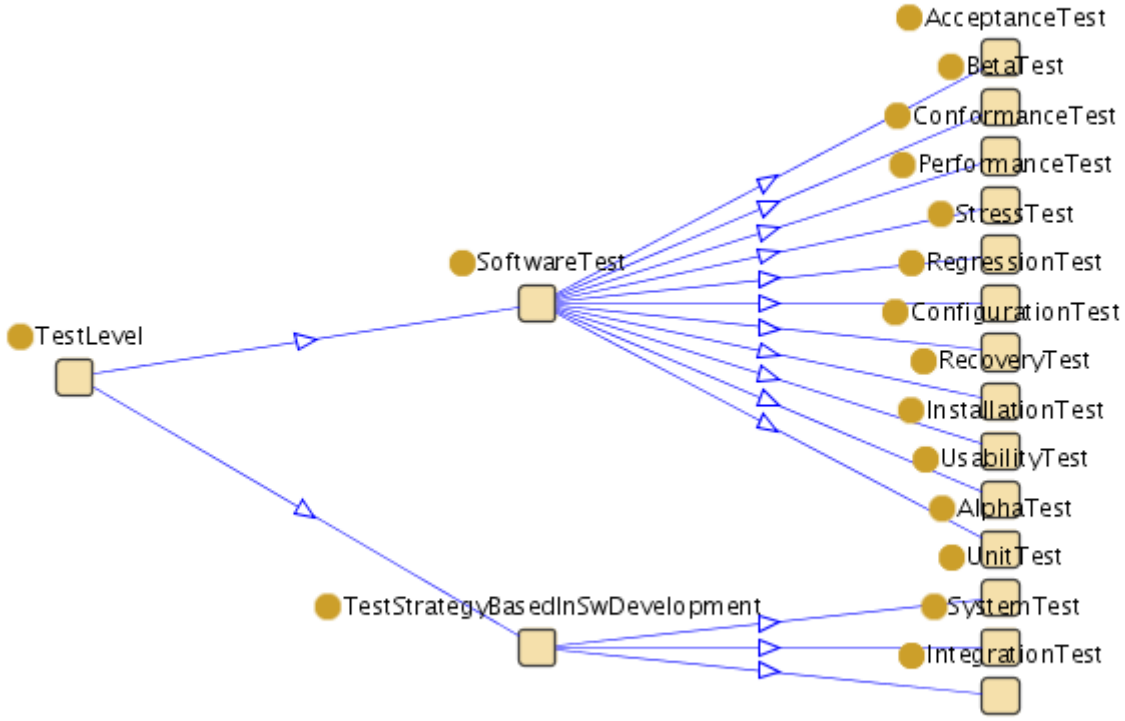

Figura 5.12: *TestLevel* e suas subclasses

- ***IntegrationTest:*** classe dos testes de integração. Trata-se de um processo de verificação entre a interação dos componentes de *software*.
- ***SystemTest:*** classe de testes de sistema. Muitas das falhas funcionais são identificadas ao longo dos testes de unidade e integração. O objetivo desta estratégia de teste é verificar requisitos não funcionais como segurança, velocidade, interfaces com aplicações externas ou com plataformas como hardware e sistema operacional.
- ***UnitTest:*** classe dos testes unitários. Esta estratégia de teste verifica a funcionalidade isolada de um *software* ou de parte dele.

- ***SoftwareTest:*** classe dos tipos de teste de *software*. Condições ou propriedades específicas do software determinam o tipo de teste que será aplicado.
  Condição Necessária:

  (i) *SoftwareTest* é subclasse de *TestGuide*.

$$\texttt{SoftwareTest} \sqsubseteq \texttt{TestLevel}$$

(ii) Indivíduos da classe *SoftwareTest* só relacionam-se através da propriedade *canBeCombinedWith* com indivíduos da classe *TestStrategyBaseInSwDevelopment*.

$$\texttt{SoftwareTest} \sqsubseteq \forall\ \texttt{canBeCombinedWith}\ .\ \texttt{TestStrategyBaseInSwDevelopment}$$

Disjunção - *A classe TestingObjetive é disjunta de TestStrategyBaseInSwDevelopment já que estas classes não compartilham instâncias*

$$\texttt{SoftwareTest} \sqsubseteq \neg\ \texttt{TestStrategyBaseInSwDevelopment}$$

- ***TestMeasurementMethod:*** esta classe representa os métodos de mensuração do teste. Uma avaliação sobre o teste pode ajudar na otimização do planejamento e a execução dos teste.
Condição Necessária:

(i) *TestMeasurementMethod* é subclasse de *SoftwareTestDomainConcept*.

$$\texttt{TestMeasurementMethod} \sqsubseteq \texttt{SoftwareTestDomainConcept}$$

(ii) Indivíduos da classe *TestMeasurementMethod* só relacionam-se através da propriedade *isTestMeasurementMethodOf* com indivíduos da classe *SoftwareTestProcess*.

$$\texttt{TestMeasurementMethod} \sqsubseteq \begin{pmatrix} \forall\ \texttt{isTestMeasurementMethodOf}\ . \\ \texttt{SoftwareTestProcess} \end{pmatrix}$$

(iii) Indivíduos da classe *TestMeasurementMethod* relacionam-se através da propriedade *isTestMeasurementMethodOf* com pelo menos um indivíduo da classe *SoftwareTestProcess*.

$$\texttt{TestMeasurementMethod} \sqsubseteq \begin{pmatrix} \exists\ \texttt{isTestMeasurementMethodOf}\ . \\ \texttt{SoftwareTestProcess} \end{pmatrix}$$

A classe *TestMeasurementMethod* possui duas subclasses disjuntas:

- ***ProgramDimensional:*** esta classe representa o método de mensuração do teste baseado em características do *software* como a complexidade, o tamanho do programa exercitado pelo teste, que pode ser medido pelo total de linhas de código ou pontos por função, entre outras características. Esta classe possui três subclasses disjuntas. São elas:
  - ***FaultDensityEvaluation:*** classe que representa a relação entre falhas encontradas durante o teste por tamanho do programa.
  - ***ReliabilityEvaluation:*** classe que representa o total de falhas que se opõem a credibilidade do *software* definida, por exemplo, por indicadores de qualidade, segurança e corretude.
  - ***ReliabilityGrowthModelEvaluation:*** classe que representa a avaliação das falhas encontradas pelo processo de teste em relação ao modelo de crescimento da credibilidade do *software*.

  - ***ProgramDimensional:*** esta classe representa o método de mensuração baseado no tamanho do teste realizado. Esta classe possui duas subclasses disjuntas. São elas:
  - ***TestCoverageEvaluation:*** esta classe representa a avaliação do teste segundo a proporção entre elementos cobertos pelo teste e o número total de elementos

do *software*, como por exemplo, classes, funções, módulos, etc.

* ***FaultSeedingEvalution:*** alguns defeitos são introduzidos propositalmente no *software* antes do teste. Quando os testes são executados, esses defeitos podem ser revelados. Dependendo de quais e quantos defeitos introduzidos propositalmente forem identificados, a efetividade do teste pode ser avaliada.

- ***TestPattern:*** esta classe representa os padrões de teste. Assim como os padrões de projeto, os padrões de teste são desenvolvidos com a finalidade de facilitar o reuso para projetos que exigem processo similar de teste.
Condição Necessária:

  (i) *TestPattern* é subclasse de *SoftwareTestDomainConcept*.

$$\texttt{TestPattern} \sqsubseteq \texttt{SoftwareTestDomainConcept}$$

  (ii) Indivíduos da classe *TestPattern* só relacionam-se através da propriedade *isTestPatternOf* com indivíduos da classe *SoftwareTestProcess*.

$$\texttt{TestPattern} \sqsubseteq \forall\ \texttt{isTestPatternOf}\ .\ \texttt{SoftwareTestProcess}$$

  (iii) Indivíduos da classe *TestPattern* relacionam-se através da propriedade *isTestPatternOf* com pelo menos um indivíduo da classe *SoftwareTestProcess*.

$$\texttt{TestPattern} \sqsubseteq \exists\ \texttt{isTestPatternOf}\ .\ \texttt{SoftwareTestProcess}$$

  Disjunção - *A classe TestPattern é disjunta de TestActivity, TestDocumentation, SoftwareTestProcess e TestGuide já que estas classes não compartilham instâncias*

$$\texttt{TestLevel} \sqsubseteq \begin{pmatrix} \neg\ \texttt{TestActivity} \\ \neg\ \texttt{TestDocumentation} \\ \neg\ \texttt{SoftwareTestProcess} \\ \neg\ \texttt{TestGuide} \end{pmatrix}$$

- ***TestTechnique:*** esta classe representa as técnicas de teste que auxiliam na identificação de incorretudes no *software*. O princípio de uma técnica é ser o mais sistemática possível e identificar um conjunto representativo em meio ao ambiente do sistema para ser exercitado pelo teste. Às vezes, as técnicas de teste são confundidas com os tipos de teste. Uma forma de diferenciar é que as técnicas devem ser vistas como apoio para ajudar a garantir a realização dos tipos de teste.
Condição Necessária:

  (i) *TestTechnique* é subclasse de *SoftwareTestDomainConcept*.

$$\texttt{TestTechnique} \sqsubseteq \texttt{SoftwareTestDomainConcept}$$

  (ii) Indivíduos da classe *TestTechnique* só relacionam-se através da propriedade *isTestTechniqueOf* com indivíduos da classe *SoftwareTestProcess*.

$$\texttt{TestTechnique} \sqsubseteq \forall\ \texttt{isTestTechniqueOf}\ .\ \texttt{SoftwareTestProcess}$$

(iii) Indivíduos da classe *TestTechnique* relacionam-se através da propriedade *isTestTechniqueOf* com pelo menos um indivíduo da classe *SoftwareTestProcess*.

$$\texttt{TestTechnique} \sqsubseteq \exists\ \texttt{isTestTechniqueOf}\ .\ \texttt{SoftwareTestProcess}$$

A Figura 5.13 ilustra as subclasses desta classe. Disjunção - *A classe TestTechnique é*

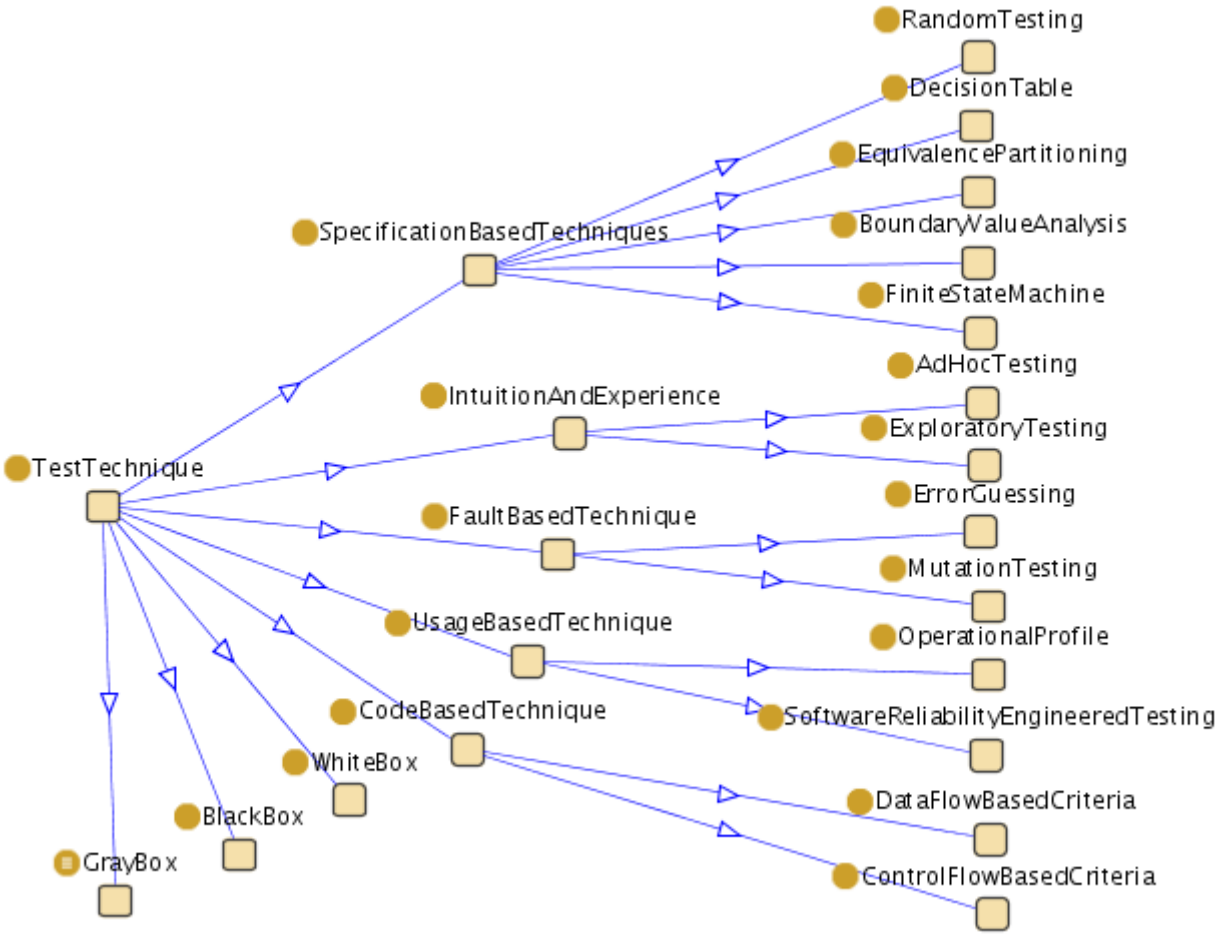

Figura 5.13: *TestTechnique* e suas subclasses

disjunta de TestActivity, TestFundamental, TestMeasurementMethod, SoftwareTestProcess e TestGuide já que não compartilha instâncias com as mesmas

$$\texttt{TestTechnique} \sqsubseteq \begin{pmatrix} \neg\ \texttt{TestActivity} \\ \neg\ \texttt{TestFundamental} \\ \neg\ \texttt{TestMeasurementMethod} \\ \neg\ \texttt{SoftwareTestProcess} \\ \neg\ \texttt{TestGuide} \end{pmatrix}$$

- ***TestTool:*** esta classe representa ferramentas de *softwares* que auxiliam o processo de teste.

Condição Necessária:

(i) *TestTool* é subclasse de *SoftwareTestDomainConcept*.

$$\texttt{TestTool} \sqsubseteq \texttt{SoftwareTestDomainConcept}$$

(ii) *TestTool* é subclasse de *OSOnto:Software*.

$$\texttt{TestTool} \sqsubseteq \texttt{OSOnto} : \texttt{Software}$$

(iii) Indivíduos da classe *TestTool* só relacionam-se através da propriedade *isTestToolUsedIn* com indivíduos da classe *SoftwareTestProcess*.

$$\texttt{TestTool} \sqsubseteq \forall\ \texttt{isTestToolUsedIn}\ .\ \texttt{SoftwareTestProcess}$$

As instâncias desta classe, conforme a Figura 5.14, representam ferramentas disponíveis no LTP para teste do Linux. A classe *TestTool* possui *TestScript* como subclasse. Os *scripts* de teste representam um conjunto de instruções geralmente armazenadas em um arquivo que devem ser interpretados linha a linha em tempo real para sua execução. A Figura 5.14 também apresenta as instâncias da classe *TestScript* no LTP.

5.3 Considerações Finais

Este Capítulo apresentou a ontologia SwTO^I (*Software Test Ontology Integrated*) que importa a OSOnto (*Operating System Ontology*) com o objetivo de representar parte do conhecimento referente ao domínio de teste do Linux apesar do escopo da SwTO^I ser mais amplo, permitindo desta forma que o processo de teste de outros sistemas também possam ser representados. O código-fonte desta onologia bem como o OWL DOC estão disponíveis em [31].

Várias teorias são formalizadas por intermédio de ontologias no entanto, poucas ontologias são efetivamente utilizadas por sistemas de informação. Isso se deve ao fato de os sistemas não tratarem e representarem o mundo real exatemente como ele é, mas da maneira mais eficiente para a realização de suas tarefas. Em alguns casos a maneira mais eficiente para os sistemas implica no uso de processamento automatizado e uma ontologia quanto mais formal mais própria ao processamento

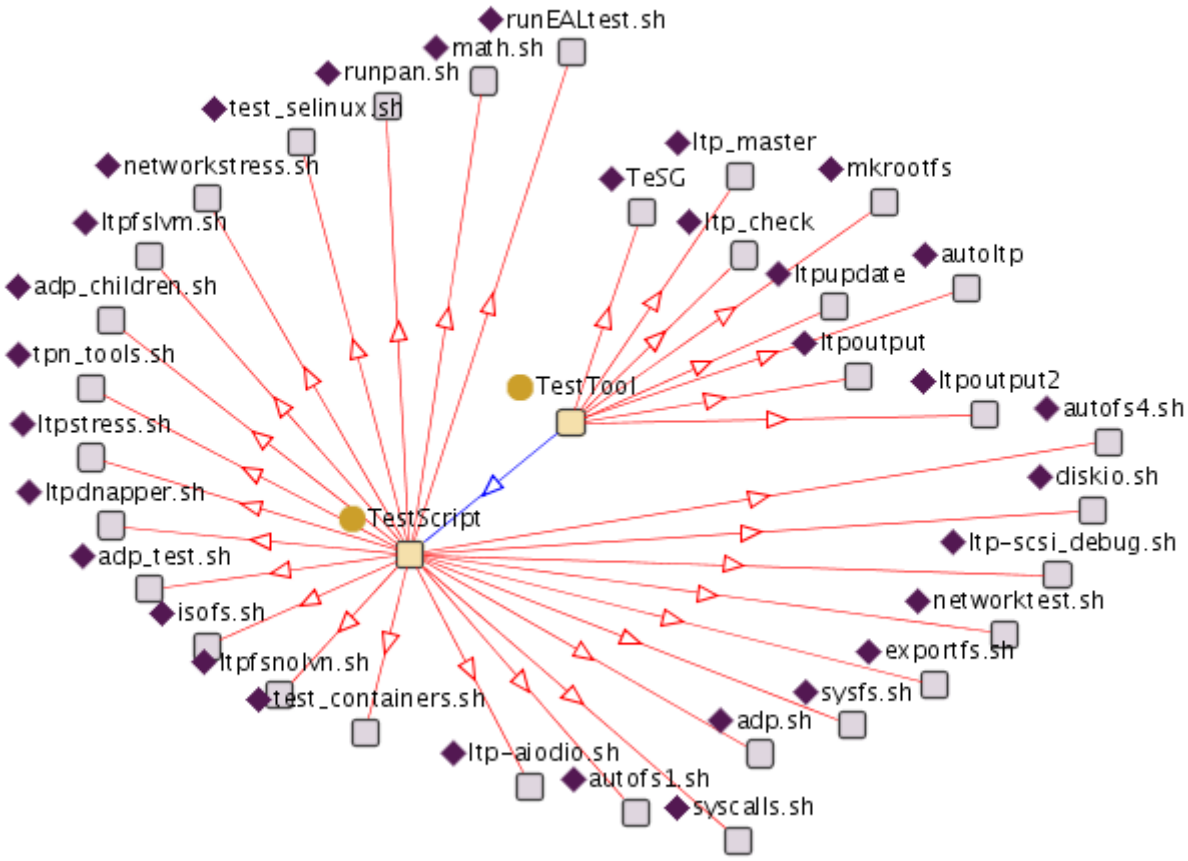

Figura 5.14: Instâncias da classe *TestTool* e *TestScript*

automatizado. Entende-se por ontologia formal aquela que dispõe não só de conceitos, relações e atributos mas restrições sobre conceitos e relações na forma de axiomas.

O próximo Capítulo mostra a utilização da SwTO^I pela ferramenta TeSG (*Test Sequence Generator*) para geração de seqüências de teste e que benefícios foram alcançados pela TeSG com o uso da SwTO^I.

Capítulo 6

A SwTO^I e a Ferramenta TeSG

Um dos objetivos definidos no Capítulo 1 foi ilustrar o uso do conhecimento contido na ontologia SwTO^I através de ferramentas que podem se beneficiar deste formalismo. No que se refere a este objetivo, a ontologia SwTO^I pode promover:

- A **comunicação**, já que a ontologia proposta mitiga a dificuldade de compreensão gerada pela ambigüidade de termos usados no domínio de teste do Linux, através da captura e representação de termos relevantes e da identificação de sinônimos;

- A **interoperabilidade** entre ferramentas de teste já que tal ontologia pode servir como uma interlíngua, ou protocolo de conhecimento entre elas.

- A **engenharia de sistemas**, já que a ontologia pode apoiar o projeto e desenvolvimento de sistemas que tenham relação com o domínio de teste do Linux.

Foram esses benefícios que motivaram a utilização da SwTO^I pela ferramenta *Test Sequence Generator*[1] (TeSG) [7] que faz uso da SwTO^I para geração de seqüências de teste.

Este Capítulo destaca a aplicação prática da ontologia SwTO^I. Para tanto, foi adotada a seguinte organização. A Seção 6.1 apresenta a TeSG. A Seção 6.2 destaca

[1] A *Test Sequence Generator* foi desenvolvida por um projeto de mestrado paralelo a este.

os resultados e benefícios obtidos pela TeSG com a utilização da SwTO^I. Finalmente, a Seção 6.3 apresenta algumas considerações sobre a aplicação da SwTO^I.

6.1 A Ferramenta TeSG

À medida que os requisitos são elicitados, o analista pode criar um conjunto de cenários que identifica uma linha de uso para o sistema a ser construído. Os cenários, freqüentemente chamados de *casos de uso*, fornecem uma descrição de como o sistema será usado. Esclarecer o que é um caso de uso e suas características é de fundamental importância na discussão da TeSG já que esta ferramenta tem a finalidade de gerar seqüências de teste a partir de casos de uso narrativos especificados para um sistema.

Para criar um caso de uso, o analista deve primeiro identificar os diferentes tipos de pessoas e/ou dispositivos (conhecidos como *atores*) que usam o sistema. Um ator, na verdade, é qualquer coisa que se comunica com o sistema e que não faz parte dele. Em geral, um caso de uso descreve o papel de um ator à medida que ocorre a interação com o sistema [27].

Requisitos de *software* podem ser documentados por casos de uso e de posse das funções para as quais o *software* foi projetado, testes podem ser criados. Esta abordagem de teste é conhecida como *funcional* ou teste *caixa-preta*. Um teste caixa-preta procura demonstrar que as funções de um *software* estão operacionais, que a entrada é adequadamente aceita, a saída é corretamente produzida e que a integridade da informação externa (por exemplo, uma base de dados) é mantida. Um teste caixa-preta examina aspectos fundamentais do sistema, se preocupando pouco com a estrutura lógica interna do *software*, para a qual existe a abordagem *caixa-branca*. Um teste caixa-branca é baseado num exame rigoroso do detalhe procedimental. Caminhos lógicos internos ao *software* são testados, exercitando conjuntos específicos de condições e/ou ciclos.

À primeira vista poderia parecer que um teste caixa-branca bastante rigoroso levaria a *softwares* corretos. Tudo o que precisaríamos seria definir todos os caminhos lógicos, desenvolver casos de teste para exercitá-los e avaliar os resultados. Infelizmente, um teste completo para um *software* pode se tornar inviável devido o número de possíveis caminhos lógicos. Para contornar o problema, o que se costuma fazer é selecionar um número limitado de caminhos lógicos importantes e exercitá-los.

Uma outra situação que pode acontecer é o *software* passar no teste caixa-branca e ser reprovado no teste caixa-preta. Do ponto de vista implementacional a lógica interna dos componentes de *software* funcionam conforme o esperado mas o *software* não atende os requisitos do cliente. Nestes casos o *software* pode ser reprovado no teste caixa-preta por conter erros de função, comportamento e/ou desempenho revelados com o exercício dos domínios de entrada e saída do *software*. Este tipo de falha pode resultar em um alto custo de manutenção já que o erro acontece na especificação, se prolonga durante toda a codificação e só é detectado posteriormente e, nos piores casos, quem detecta o erro é o cliente após a entrega do *software*.

O método de teste proposto em [7] contempla a ferramenta TeSG e oferece uma solução para evitar este tipo de falha crítica e custosa. Para tanto, a TeSG se apoia no conhecimento representado pela SwTO^I. Mas, porque uma ontologia é utilizada pela TeSG? Porque a SwTO^I foi escolhida?

Como dito anteriormente, os casos de uso são documentos textuais, descritos em linguagem natural. Para que eles possam ser interpretados e processados computacionalmente, é necessário descrevê-los formalmente. Só que simplesmente transcrever casos de uso de uma linguagem natural para uma linguagem computacional não é suficiente. Para se gerar seqüências de teste com probabilidade de encontrar erros é necessário unir o conhecimento do projetista referente a teoria de teste de *software* e sua relação com os casos de uso. Esta união entre representação de conhecimento e conjuntos que instancializem este conhecimento pode ser obtida por intermédio de

ontologias formais. Este benefício fez com que a TeSG considerasse ontologia como um formalismo adequado para ser utilizado na solução do problema.

A pesquisa realizada em [7] aponta a SwTO^I como a ontologia formal mais relevante para o domínio de aplicação da TeSG, por isso esta ontologia foi escolhida.

A Figura 6.1 ilustra o processo em alto nível no qual a TeSG faz uso da SwTO^I para geração de seqüências de teste.

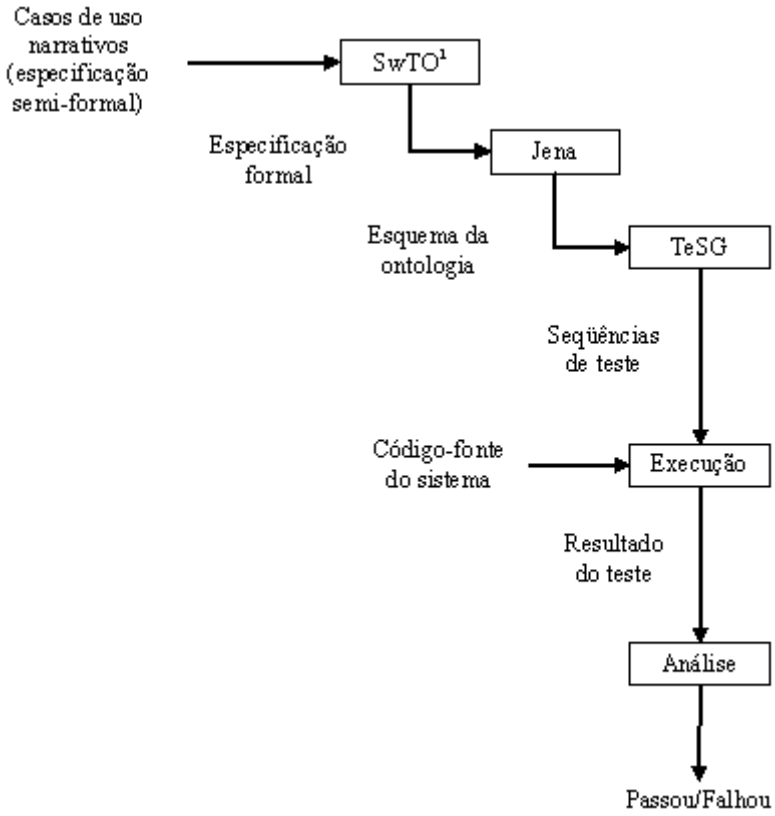

Figura 6.1: Visão geral do fluxo da TeSG

O *primeiro estágio* do fluxo é representado pelos casos de uso narrativos de um sistema como entrada para SwTO^I. Os casos de uso passam a ser instâncias da classe *UseCase* discutida na seção 6.1.1.

O *segundo estágio* do fluxo é representado pela especificação formal da SwTO^I como entrada para o *framework* Jena[2]. Do código-fonte da SwTO^I, o Jena gera um

[2]http://jena.sourceforge.net

esquema para que aplicações desenvolvidas em Java possam manipular a TBox e a ABox da ontologia mais facilmente.

O *terceiro estágio* do fluxo é representado pelo esquema gerado a partir do Jena como entrada para a TeSG, desenvolvida em Java, esta ferramenta faz uso de um algoritmo genético como abordagem heurística para combinar eventos entre casos de uso relacionados. Segundo [35], em *algoritmos genéticos*, uma população de possíveis soluções para o problema em questão evolui de acordo com operadores probabilísticos concebidos a partir de metáforas biológicas, de modo que há uma tendência de que, na média, os indivíduos representem soluções cada vez melhores à medida que o processo evolutivo continua.

Um método heurístico foi utilizado neste passo já que a abordagem de teste descrita em [7] se concentra em um procedimento combinatorial de teste. A geração de seqüências de teste tem complexidade exponencial. De acordo com o acréscimo de casos de uso e seus respectivos eventos como entrada para a ferramenta, a seqüência gerada pode ser inviável, possivelmente incapaz de ser diretamente executada por pessoas em tempo hábil, implementada em uma linguagem em tempo compatível com o cronograma do projeto ou processada em tempo polinomial. Por este motivo, a ferramenta procurou seguir uma abordagem heurística que leva em consideração o tamanho da solução, operadores probabilisticos, casos de uso relacionados e eventos mais relevantes com maior prioridade segundo análise do especialista porém, não descarta os eventos de prioridade mais baixa. Esses critérios evitam que uma seqüência inviável seja gerada.

As instâncias de casos de uso e suas respectivas seqüências de eventos inseridas no primerio estágio do fluxo são resgatadas do esquema gerado pelo Jena e processadas pela TeSG gerando seqüências de teste que representam a entrada para o próximo estágio do fluxo.

No *quarto estágio*, o código-fonte do sistema é submetido para execução de testes

conforme a seqüência narrativa gerada no terceiro estágio que pode ser utilizada em um teste manual ou como especificação para uma posterior implementação em qualquer linguagem de programação, viabilizando desta forma a automação dos testes.

O *quinto estágio* do fluxo é representado pela análise dos resultados do teste que aponta se o *software* passou ou falhou.

Como foi discutido no Capítulo 5, a SwTO^I importa a OSOnto (discutida em detalhes no Apêndice A). A próxima seção detalha os conceitos presentes na OSOnto que são efetivamente utilizados pela TeSG através da SwTO^I para geração de seqüências de teste.

6.1.1 Conhecimento Formal da OSOnto Usado pela TeSG para Geração de Seqüências de Teste

A Figura 6.2 apresenta parte da estrutura hierarquica da ontologia OSOnto. A classe de mais alto nível neste fragmento da taxonomia é *SoftwareDocumentation* que representa o conjunto das documentações de *software*. São subclasses de *SoftwareDocumentation* as classes *DesignDocumentation*, *MarketingDocumentation*, *TechnicalDocumentation* e *UserDocumentation*. No momento, somente as subclasses de *DesignDocumentation* relevantes à TeSG serão discutidas com o objetivo de esclarecer como o conhecimento formalizado é efetivamente utilizado pela TeSG.

A classe *UseCase*

A Figura 6.3, extraída do ambiente Protégé, ilustra as condições lógicas da classe *UseCase*. Por estas condições lógica é possivel observar que os casos de uso:

- Também são indivíduos de uma de suas subclasses, ou seja, os casos de uso ou são expandidos ou são de alto nível já que as classes *Expanded* e *HighLevel* são disjuntas. Os casos de uso expandidos são aqueles que possuem uma sequência de eventos. Os casos de uso de alto nível são aqueles que não possuem sequências de eventos. A TeSG considera na geração de seqüências de teste os casos de uso *Expanded*. Por serem detalhados, é possível mesclar eventos entre casos de uso relacionados e obter uma sequência de teste. A Figura 6.4 ilustra esta relação;
- *UseCase* é subclasse de *DesignDocumentation*

Figura 6.2: A classe *SoftwareDocumentation* e suas subclasses

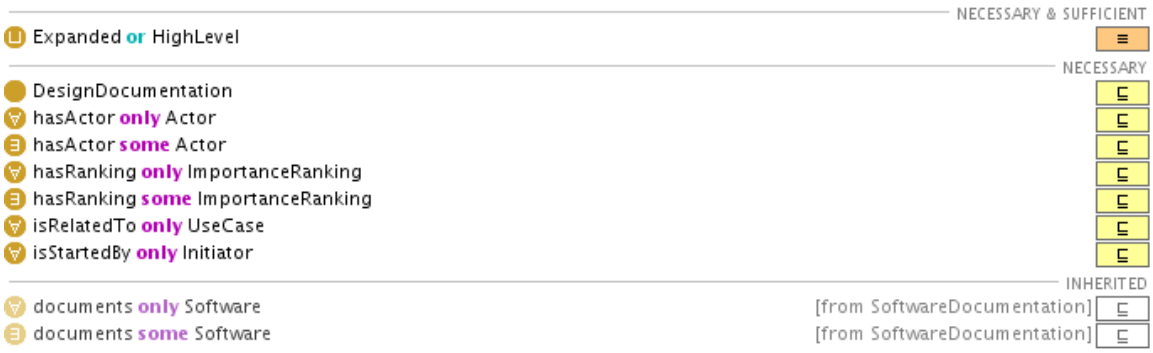

Figura 6.3: Descrição lógica da classe *UseCase*

- Indivíduos da classe *UseCase* só relacionam-se através da propriedade *hasActor* com indivíduos da classe *Actor* e com pelo menos um indivíduo desta classe através da propriedade *hasActor*;
- São categorizados de acordo com seu grau de importância, ou seja, são primários ou secundários. Casos de uso primários são aqueles que ocorrem com freqüência. Casos de uso secundários são aqueles menos importantes ou raros de acontecerem. Os casos de uso só relacionam-se através da propriedade *hasRanking* com indivíduos da classe *ImportanceRanking* e com pelo menos um indivíduo desta classe através da propriedade *hasRanking*. A Figura 6.5 ilustra essa relação.

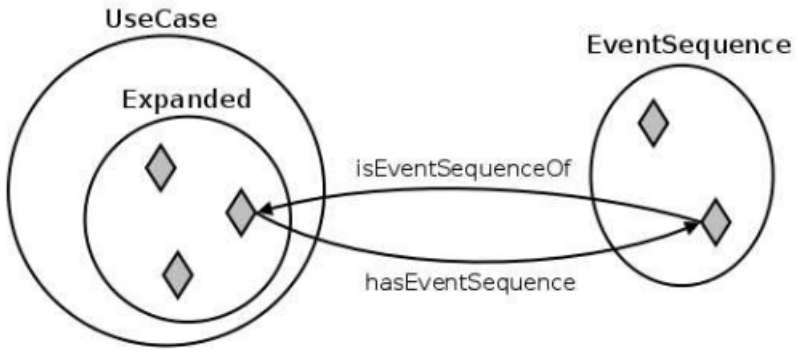

Figura 6.4: Relacionamento entre indivíduos da classe *Expanded* e *EventSequence* [7].

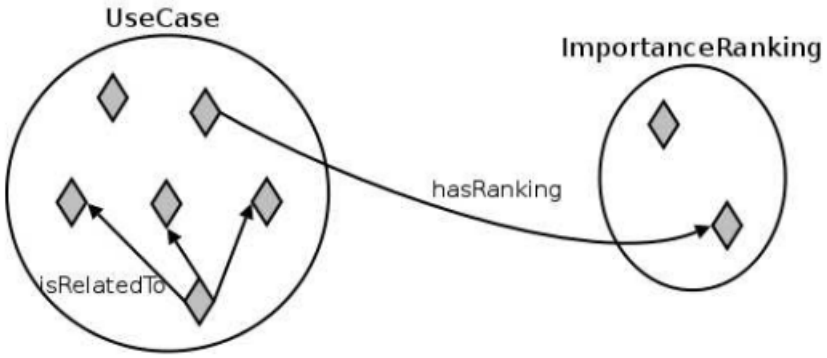

Figura 6.5: Relacionamento entre indivíduos da classe *UseCase* e *ImportanceRanking* [7].

- Um caso de uso pode ter relação com outros casos de uso (ilustrado na Figura 6.5). Neste tabalho, não foram diferenciados ou classificados os tipos de relações entre os casos de uso por uma questão de projeto. A relação entre os casos de uso pode ser detalhada em trabalhos futuros como *extensão*, que representa a extensão de uma funcionalidade de um caso de uso para incorporar procedimentos opcionais; *inclusão*, representa a inclusão de um outro caso de uso; e *generalização*, que permite aos casos de uso suportarem herança, isto é, permite a criação de classes de uso que herdem atributos e operações de outras classes. A idéia de relação usada neste trabaho, baseia-se na função representada pelo caso de uso, mesmo que ele não extenda, inclua ou generalize outro caso de uso. Por exemplo: dado um caso de uso A que representa uma função do sistem e que manipula um recurso deste mesmo sistema e um caso de uso B que representa uma outra função do mesmo módulo do caso de uso A e que manipula o mesmo recurso, então dizemos que os casos de uso são relacionados. Este tipo de relacinamento visa encontrar erros duarante a execução do sistema, através da combinação dos eventos presentes nos casos de uso relacionados.

  Esta relação estabelecida entre os casos de uso é feita pelo especialista em teste que usa a sua experiência para determinar quais são os casos de uso relacinados. A TeSG atualmente não faz uso dos tipos de relação entre casos de uso para gerar as seqüência de teste. Basta para a TeSG saber quais os casos de uso relacionados.

- Um caso de uso é iniciado por um ator iniciador;

A Figura 6.6 ilustra as propriedades que descrevem a classe *UseCase*. Por esta

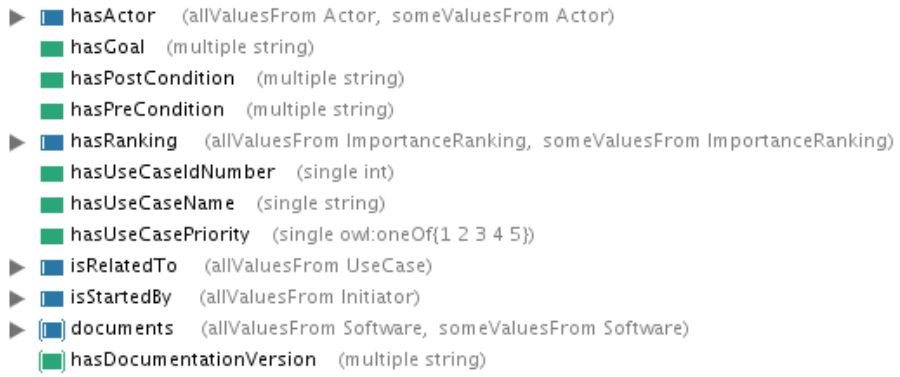

Figura 6.6: Propriedades que descrevem a classe *UseCase*

descrição é possível observar que:

- As propriedade *hasGoal, hasPostCondition, hasUseCaseIdNumber, hasUseCaseName, hasUseCasePriority* e *hasDocumentationVersion* são propriedades de tipo de dados.
- As propriedades *hasActor, hasRanking, isRelatedTo, isStartedBy* e *documents* são propriedades objeto.
- A propriedade *hasPostCondition* e *hasPreCondition* são muito importantes para a geração de seqüência de teste. A TeSG avalia se um caso de uso tem pré ou pós-condição. Caso tenha, esta informação vai compor a sequência de teste.
- Quem define a propriedade de um caso de uso através da propriedade *hasUseCasePriority* é o especialista do domínio onde 5 é a prioridade máxima é 1 representa a prioridade mínima.

A classe *Actor*

A Figura 6.7 ilustra as condições lógica da classe *Actor*. Por esta condições é

Figura 6.7: Descrição lógica da classe *Actor*

possivel observar que:

- Todos os atores são indivíduos de uma de suas subclasses já elas são disjuntas, ou seja, um ator pode ser um *hardware*, *software*, ou uma pessoa.
- Indivíduos da classe *Actor* só relacionam-se através da propriedade *isActorOf* com pelo menos um indivíduo da classe *UseCase*;
- Um ator representa um comportamento externo ao sistema. Indivíduos da classe *Actor* só relacionam-se através da propriedade *isDelimitedBy* com indivíduos da classe *ExternalBehavior*

Dentro dos atores que participam de um caso de uso existe uma classe especial que são os atores inicializadores. Estes se diferem dos demais por serem responsávies em inicializar um caso de uso. A Figura 6.8 ilustra instâncias da classe *Actor* e de sua subclasse, bem como a relação dessas instâncias com a das classes *ExternalBehaviour* e *UseCase*.

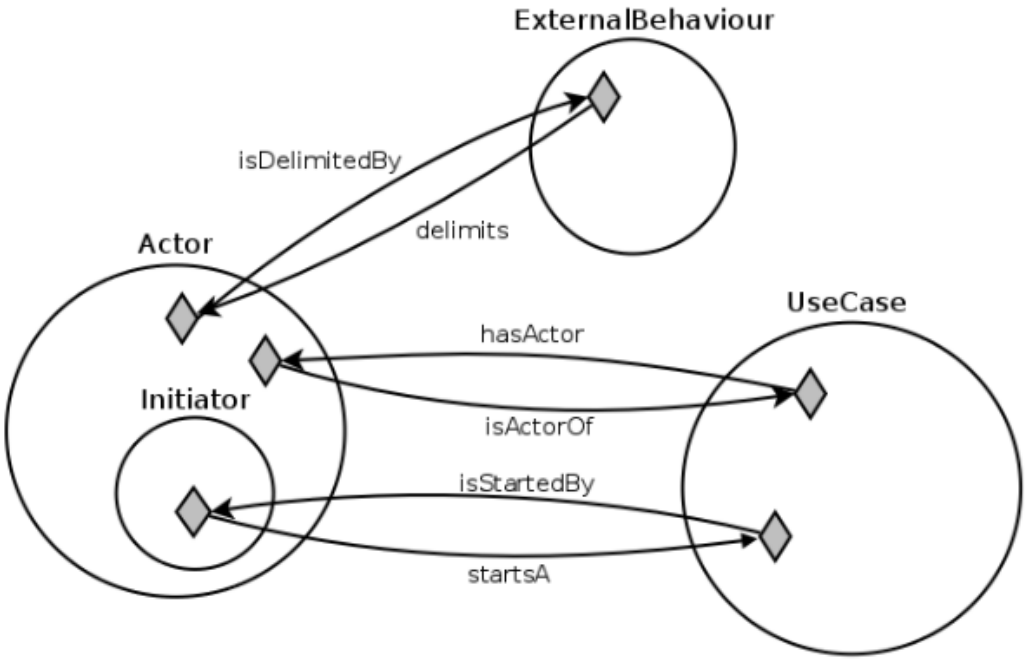

Figura 6.8: Relacionamento entre instâncias da classe *Actor, Initiator, ExternalBehavior* e *UseCase* [7]

A Figura 6.9 ilustra as propriedades que descrevem a classe *Actor*. Por esta

Figura 6.9: Propriedades que descrevem a classe *Actor*

descrição é possível observar que:

- As propriedades *hasActorDescription* e *hasActorName* são propriedades de tipo de dados.
- As propriedades *isActorOf* e *isDelimitedBy* são propriedades objeto.

A classe *Event*

A Figura 6.10 ilustra a descrição lógica da classe *Event*. Por esta descrição é

Figura 6.10: Descrição lógica da classe *Event*

possivel observar que:

- Indivíduos da classe *Event* só relacinam-se através da propriedade *isEventOf* com pelo menos um indivíduo da classe *EventSequence*

A Figura 6.11 ilustra as propriedades que descrevem a classe *Event*. Por esta des-

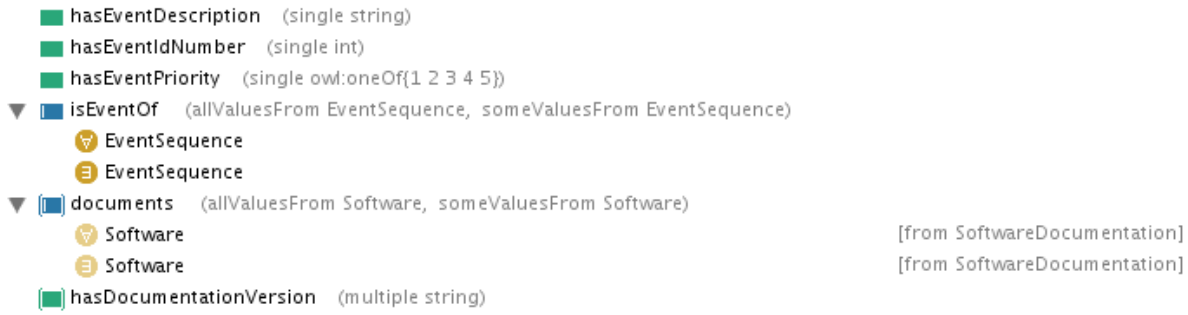

Figura 6.11: Propriedades que descrevem a classe *Event*

crição é possível observar que:

- As propriedades *hasEventDescription, hasEventIdNumber* e *hasEventPriority* são de tipo de dados.
- *isEventOf* é uma propriedade objeto inversa a *hasEvent*. A Figura 6.12 ilustra a relação entre instâncias através desta propriedade.
- A propriedade de tipo de dados *hasEventPriority* é de fundamental importância para a TeSG. Os valores para a prioridade de um evento foram pré-definidos na ontologia e variam de 1 a 5, onde 5 representa prioridade máxima e 1 representa prioridade mínima. O algoritmo genético interpreta a soma das prioridades dos eventos que formam uma seqüência contida em um caso de uso expandido como uma função *fitness* [7]. A função *fitness* é um dos indicadores capaz de dizer se a sequência teste gerada tende a ser boa ou não a partir da combinação dos eventos.

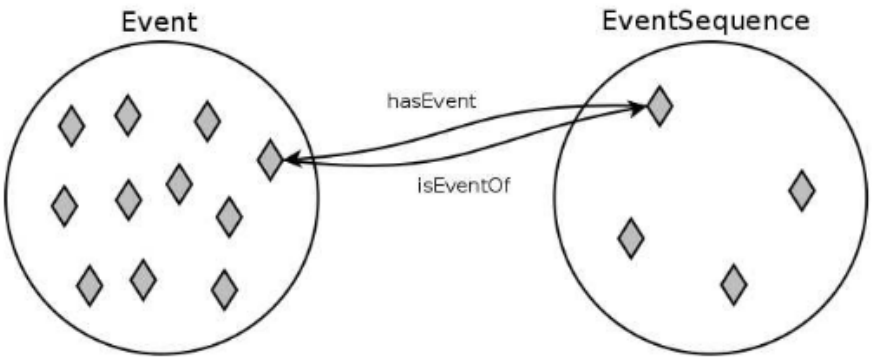

Figura 6.12: Relacionamento entre instâncias da classe *Event* e *EventSequence* [7]

A classe *Event* possui quatro subclasses a saber, *FirstEvent*, *IntermediateEvent*, *LastEvent* e *UnitEvent*. Essas classes não foram declaradas como disjuntas já que podem compartilhar instâncias. Por exemplo, um evento pode ser definido como primeiro evento em uma seqüência A, e na seqüência B este mesmo evento pode ser intermediário. A seguir, as subclasses de *Event* serão detalhadas.

A classe *FirstEvent*

A Figura 6.13 ilustra a descrição lógica da classe *FirstEvent*. Por esta descrição

Figura 6.13: Descrição lógica da classe *FirstEvent*

é possivel observar que:

- O primeiro evento de uma seqüência é aquele que não tem um evento antecessor mas tem exatamente um evento sucessor.

A classe *IntermediateEvent*

A Figura 6.14 ilustra a descrição lógica da classe *IntermediateEvent*. Por esta descrição é possivel observar que:

- Um evento intermediário é aquele que tem exatamente um evento antecessor e um evento sucessor.

Figura 6.14: Descrição lógica da classe *IntermediateEvent*

A classe *LastEvent*

A Figura 6.15 ilustra a descrição lógica da classe *LastEvent*. Por esta descrição

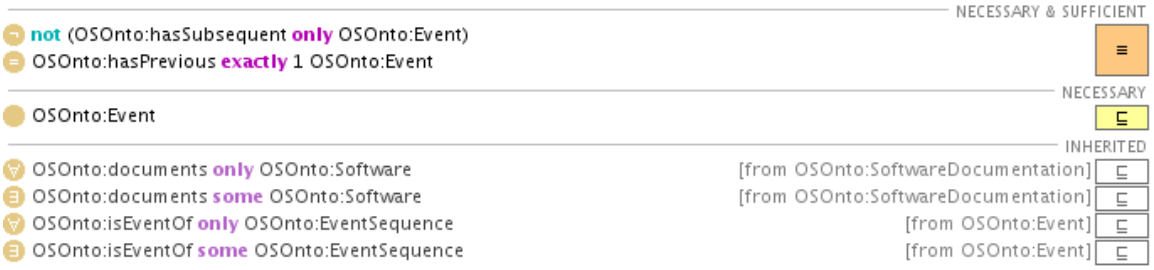

Figura 6.15: Descrição lógica da classe *LastEvent*

é possivel observar que:

- O último evento de uma seqüência é aquele que tem exatamente um evento antecessor e não tem evento sucessor.

A classe *UnitEvent*

A Figura 6.16 ilustra a descrição lógica da classe *UnitEvent*. Por esta descrição

Figura 6.16: Descrição lógica da classe *UnitEvent*

é possivel observar que:

- Um evento unitário é aquele que não tem nem antecessor nem sucessor.

A Figura 6.17 ilustra as subclasses de *Event* e o relacionamento entre instâncias.

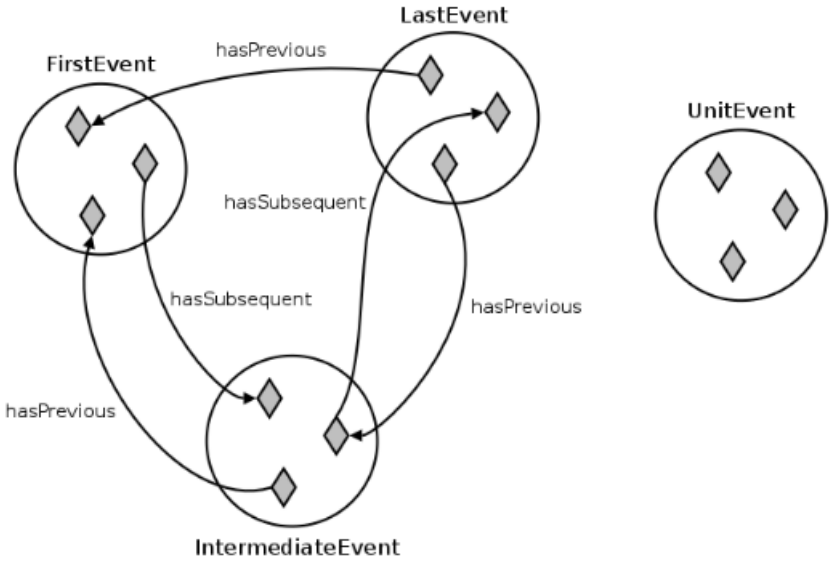

Figura 6.17: As subclasses de *Event* e o relacionamento entre instâncias [7]

A classe *EventSequence*

A Figura 6.18 ilustra a descrição lógica da classe *EventSequence*. Por esta des-

Figura 6.18: Descrição lógica da classe *EventSequence*

crição é possivel observar que:

- Uma seqüência de eventos será típica ou alternativa;
- Uma seqüência de eventos só relaciona-se através da propriedade *hasEvent* com pelo menos um indivíduo da classe *Event*.

6.2 Resultados Obtidos pela TeSG com a Utilização da SwTO^I

Os experimentos de utilização da SwTO^I pela TeSG para geração de seqüências de teste estão descritos em [7]. Basicamente dois experimentos foram realizados. O

primeiro experimento foi gerar seqüências de teste para o *Virtual File System*(VFS) do Linux e o segundo foi gerar seqüências de teste para o SiGIPós (Sistema Gestor de Informação do Programa de Pós-Graduação em Informática) [6].

6.2.1 Seqüências de Teste para o VFS do Linux

Em alto nível, a Figura 6.19 apresenta as principais etapas do primerio experimento que iniciou com o estudo dos casos de teste presentes no LTP (*Linux Test Project*) para o VFS. Do código-fonte dos casos de teste (implementados na linguagem C) foi feito um procedimento de engenharia reversa para extrair a especificação e identificar atores, casos de uso expandidos com seqüências de eventos típicas e alternativas, definição da prioridade dos casos de uso e definição da prioridade dos eventos. De posse dos casos de uso narrativos, eles foram formalizados através da SwTO^I e passaram a ser instâncias para a teoria representada. O ambiente Protégé e raciocinador Racer foram utilizados no procedimento de instanciação dos casos de uso e conceitos associados para garantir a consistência. Posteriormente a ferramenta TeSG, por intermédio do *framework* Jena, gerou o esquema da SwTO^I contendo toda a formalização do domínio as instâncias e todas as suas características necessárias para o experimento. Os indicadores do algoritmo genético também foram ajustados de forma a gerar um processo combinatorial eficiente para a população selecionada. Com todas as variáveis de entrada devidamente disponíveis e configuradas, a TeSG foi utilizada para gerar as seqüências de teste narrativas que foram posteriormente implementadas na linguagem C. A próxima etapa do experimento foi modificar uma parte específica do código-fonte do VFS do Linux, resultando em incompatibilidades com a especificação, ou seja, *bugs*. De posse do código-fonte do VFS do Linux modificado, dos testes gerados pela TeSG e codificados na linguagem C, o próximo passo foi usar os *scripts* de teste do LTP para executar os casos de teste e obter relatórios. Estes relatórios apontaram falhas, ou seja, os *bugs* foram detectados. Com

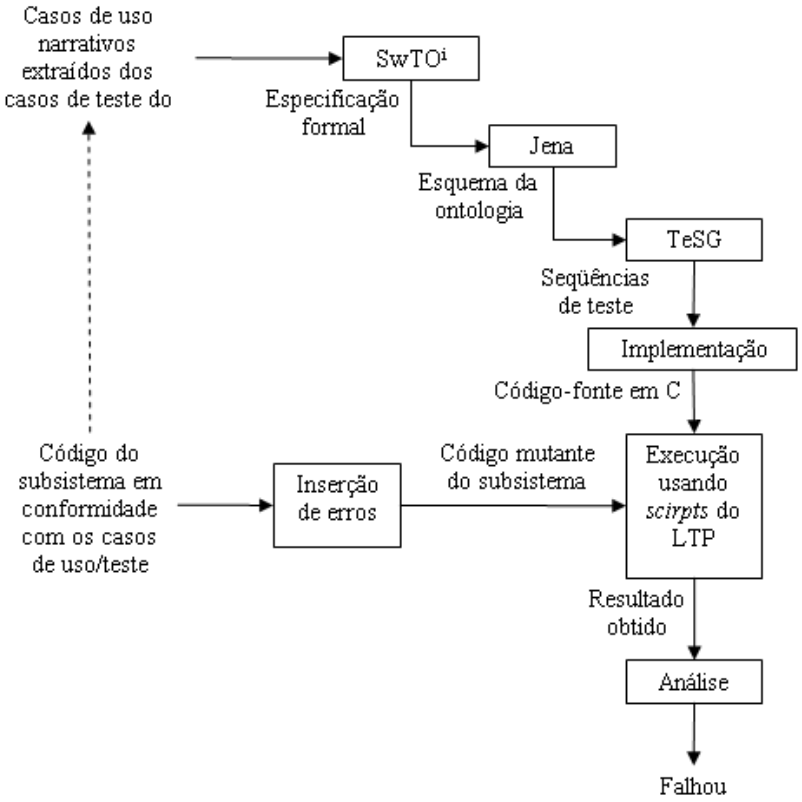

Figura 6.19: Fluxo em alto nível do primeiro experimento da TeSG usando a SwTO^I

este primeiro experimento foi possível comprovar que o conhecimento representado pela SwTO^I auxilia a TeSG na geração de seqüências de teste narrativas que pode ser implementada posteriormente.

6.2.2 Seqüências de Teste para o SiGIPós

O SiGIPós possui uma interface Web e é dividido em vários subsistemas, dentre os quais foi escolhido o subsistema Calendário para realização do experimento. Este subsistema tem por objetivo auxiliar o gerenciamento de informação sobre o calendário acadêmico.

Este segundo experimento foi mais simples, se comparado ao primerio experimento. De forma análoga ao primeiro, o segundo experimento iniciou com a formalização dos casos de uso na SwTO^I. Após este procedimento, o Racer foi usado para

verificar a consistência da ontologia. Posteriormente o Jena foi usado para gerar um novo esquema da SwTO^I, desta vez com uma ABox enriquecida com instâncias referentes ao SiGIPós. A última etapa do experimento foi gerar as seqüências de teste usando os mesmos parametros para o algoritmo genético do experimento anterior. Através deste experimento, foi possível gerar uma seqüência de teste para um *software* de domínio diferente do primeiro experimento. Isto nos informa que tanto SwTO^I quanto o método proposto em [7] podem ser usados para aplicações de diferentes domínio.

6.3 Considerações Finais

A ferramenta TeSG efetivamente incorporou a ontologia SwTO^I como parte da sua arquitetura e necessita do conhecimento contido na ontologia para gerar seqüências de teste. No ciclo de desenvolvimento da TeSG a ontologia esteve presente na fase de análise, contribuindo com conhecimento estruturado e organizado facilitando desta forma o entendimento do domínio. O desenvolvedor da ferramenta pode ter uma visão ampla dos conceitos do domínio e suas relações bem como uma visão detalhada de algumas classes e suas definições, propriedades e instâncias. Durante a fase de projeto da TeSG, a ontologia contribuiu novamente, viabilizando o reuso do conhecimento pela ferramenta. Durante as fases de teste da TeSG, a SwTO^I foi enriquecida com novas instâncias que por sua vez foram devidamente classificadas, direcionando desta forma um teste mais efetivo da própria ferramenta.

Este processo de desenvolvimento da TeSG orientado a ontologia trouxe benefícios que surpreenderam. Primeiramente o projeto da TeSG que se mostrou bem organizado, direcionado e com escopo bem definido. Isso poderia ser alcançado sem o auxílio de uma ontologia. Entretanto, o processo de análise do domínio da ferramenta se mostrou mais ágil por se apoiar em uma teoria compilada para o domínio do problema. Sem a ontologia, o desenvolvedor teria que ir em diversas

fontes bibliográficas para abstrair conceitos do domínio o que exigiria um tempo extra de pesquisa. Experimentos realizados com a TeSG em [7] comprovam que o conhecimento representado pela SwTO^I permite a geração eficiente de sequência de teste e comprova que o uso de ontologias torna possível representar informações que refletem um entendimento semântico quanto ao teste de software. O segundo ponto observado foi a qualidade final da seqüência de teste gerada pela ferramenta nos dois experimentos realizados.

Capítulo 7

Avaliação da SwTO^I

O objetivo deste Capítulo é apresentar o procedimento utilizado e os resultados obtidos na avaliação da SwTO^I.

A finalidade da avaliação é apontar a qualidade de uma ontologia. Alguns trabalhos se concentram no desenvolvimento e estudo dos métodos para tanto existem algumas ferramentas que se propõem a apoiar esta atividade, mas nenhuma é utilizada em larga escala [33].

A avaliação realizada na SwTO^I procurou seguir duas estratégias em paralelo: uma avaliação quantitativa e uma avaliação qualitativa.

A avaliação quantitativa usou como base o método descrito em [15], que também foi enriquecido neste trabalho com novos critérios como classe de maior grau e níveis da ontologia. A avaliação qualitativa usou como base o método descrito em [37], que foi enriquecido neste trabalho com a utilização de um raciocinador e métricas. Ambas as estratégias se complementam e serão descritas a seguir.

Este Capítulo está organizado da seguinte forma. Na seção 7.1 temos a avaliação quantitativa e os resultados obtidos. Na seção 7.2 temos a avaliação qualitativa e os resultados obtidos. Finalmente, a Seção 7.3 apresenta algumas considerações sobre a avaliação realizada.

7.1 Avaliação Quantitativa

Apesar das ontologias serem utilizadas para representar conhecimento dos mais variados domínios, as ontologias podem apresentar componentes em comum como conceitos, instâncias, atributos (de conceitos ou instâncias), relações que podem ser hierarquicas (todo-parte e é-um) ou não e restrições sobre conceitos e relações na forma de axiomas. A análise desses componentes que compõem uma ontologia podem revelar informações e características importantes como por exemplo, o nível de formalidade da ontologia que varia de acordo com a especificação explícita. Ontologias leves ou informais comportam conceitos, instâncias, relações e atributos. Ontologias pesadas ou formais, além de comportarem os mesmos componentes das ontologias informais, também representam restrições sobre conceitos e relações na forma de axiomas.

O método proposto por Ning Huang e Shihan Diao em [15] sugere uma avaliação baseada na estrutura da ontologia, ou seja, este método procura elicitar quantitativamente os componentes de uma ontologia e usa para isso a teoria de grafos e indicadores estatísticos. O método também procura apontar o nível de estruturação da ontologia. Uma ontologia de estrura bem organizada pode prover um fácil entendimento do domínio representado e um bom indício de aplicabilidade, integrabilidade e reusabilidade. O método também viabiliza a análise e comparação entre ontologias mesmo que elas representem domínios distintos justamente por se concentrar nos componentes ontológicos. Como não há na literatura referência de uma outra ontologia que represente o domínio de teste do Linux, procuramos através deste método, aplicar os mesmos indicadores sobre a *Pizza Ontology* [24, 14] por ser uma referência didática em muitas disciplinas de engenharia de ontologias e é bem discutida no meio acadêmico. Apesar de terem domínios diferentes, a *Pizza Ontology* e a SwTO^I foram implementadas em OWL e possuem classes, propriedades

instâncias entre outras características em comuns.

A avaliação quantitativa foi feita com base em cinco indicadores: (i) quantidade de classes nomeadas, (ii) média das propriedades P_O, (iii) níveis da ontologia quanto a relação é-um, (iv) classe de maior grau da ontologia no que se refere à relação é-um e (v) classe de maior grau da ontologia no que se refere à relação todo-parte. A análise isolada de cada um desses indicadores não leva a resultados conclusivos. É necessários observá-los em conjunto. A seguir os indicadores serão detalhados.

7.1.1 Quantidade de Classes Nomeadas

A quantidade de classes nomeadas pode fornecer um indicador do tamanho da ontologia e do domínio representado. Este indicador pode ser útil para comparar ontologias de um mesmo domínio. Digamos que as ontologias A e B representem conhecimento sobre *pizzas*. A ontologia A contém 100 classes nomeadas e a ontologia **B** contém apenas 9. Esta informação nos dá a idéia que a ontologia A é mais rica em conceitos. Entretanto, este indicador não é conclusivo pois para avaliar o nível de detalhe de uma ontologia são necessários outros indicadores que complementem a análise.

A tabela 7.1 apresenta a quantidade de classes nomeadas da SwTO^I e da *Pizza Ontology*. Apesar de serem de domínios diferentes, isso não inviabiliza a análise já que este indicador tem como objetivo elicitar, mesmo que parcialmente, o tamanho da ontologia através da quantidade de classes nomeadas.

Ontologia	Total de classes nomeadas
SwTO^I	194
Pizza	99

Tabela 7.1: Total de classes nomeadas por ontologia

7.1.2 Média das Propriedades P_O

Em OWL DL uma propriedade é uma relação binária. Neste trabalho são considerados dois tipos de propriedades: propriedades de tipo de dados (P_D) e propriedades objeto (P_O).

Uma propriedade objeto reflete uma relação entre instâncias de duas classes. Os raciocinadores realizam inferências sobre propriedades objeto como a verificação de restrições sobre propriedades e a verificação do axioma de domínio e contradomínio. Uma propriedade de tipo de dados reflete uma relação entre uma instância e um tipo *XMLSchema*. As propriedades de tipo de dados são tratada separadamente pelo raciocinador. Geralmente existe uma máquina de inferência separada para tipos de dados. Devido a premissa lógica de mundo aberto (*open world assumption*) dos raciocinadores utilizados sobre OWL, a análise da média das propriedades objeto em relação ao total de classes nomeadas nos dá uma idéia da distribuição de uma representatividade entre as instâncias das classes. A tabela 7.2 apresenta a quantidade de propriedades objeto para cada ontologia. Pelo resultado apresentado

Ontologia	Total de P_O	Total de P_D
SwTO^I	142	51
Pizza	8	0

Tabela 7.2: Total de propriedades por ontologia

na tabela 7.2 é possível observar que uma ontologia pode até não ter propriedades P_D, como é o caso da *Pizza Ontology*, mas as propriedades P_O são indicadas para uma melhor representação do domínio.

Na equação a seguir, MPO significa média de propriedades objeto da ontologia; P_O é o total de propriedades objeto; n é o total de classes nomeadas da ontologia.

$$MPO = \frac{P_O}{n}$$

A tabela 7.3 apresenta a média de cada ontologia em relação ao total de classes nome-

adas. Esta indicador pode ajudar os engenheiros de ontologia a avaliar a abundância

Ontologia	Média de P_O
SwTO^I	0,73
Pizza	0,08

Tabela 7.3: Média de P_O por ontologia

de relações entre classes.

7.1.3 Níveis da Ontologia quanto a relação é-um

Ao observar a estrutra de uma ontologia é possível fazer uma analogia com a teoria de grafos [40] onde uma classe corresponde a um vértice e a relação entre classes ("é-um"ou "todo-parte") corresponde a uma aresta. Os grafos são ricos em propriedades e muitas informações podem ser extaídas a partir da sua análise. Por exemplo:

- Um grafo pode ser direcionado ou não direcionado;
- Um vértice pode ter grau e isso pode indicar se o grafo tem vértices não conectados ou isolados;
- Se o grafo for não direcionado, ele pode formar uma árvore, ou seja, ele será acíclico e conectado.
- Várias árvores reunidas formam uma floresta.

Para uma mesma ontologia, vários grafos podem ser gerados. Se levarmos em consideração a relação "é-um"entre as classes, temos uma árvore e se uma ontologia importa a outra, temos uma floresta. A partir deste tipo de grafo é possivel identificar o nível de profundidade da ontologia, partindo do conceito de mais alto nível ou conceito raiz até chegar nos conceitos-folha.

A figura 7.1 gerada pelo *plug-in* Jambalaya ilustra a árvore de classes e relações é-um da ontologia SwTO^I. Por esta árvore é possível observar que esta ontologia

tem sete níveis desde o nó raiz até as folhas sendo que o quinto nível da raiz para as folhas é o mais denso, com maior quantidade de classes. A figura 7.2 ilustra

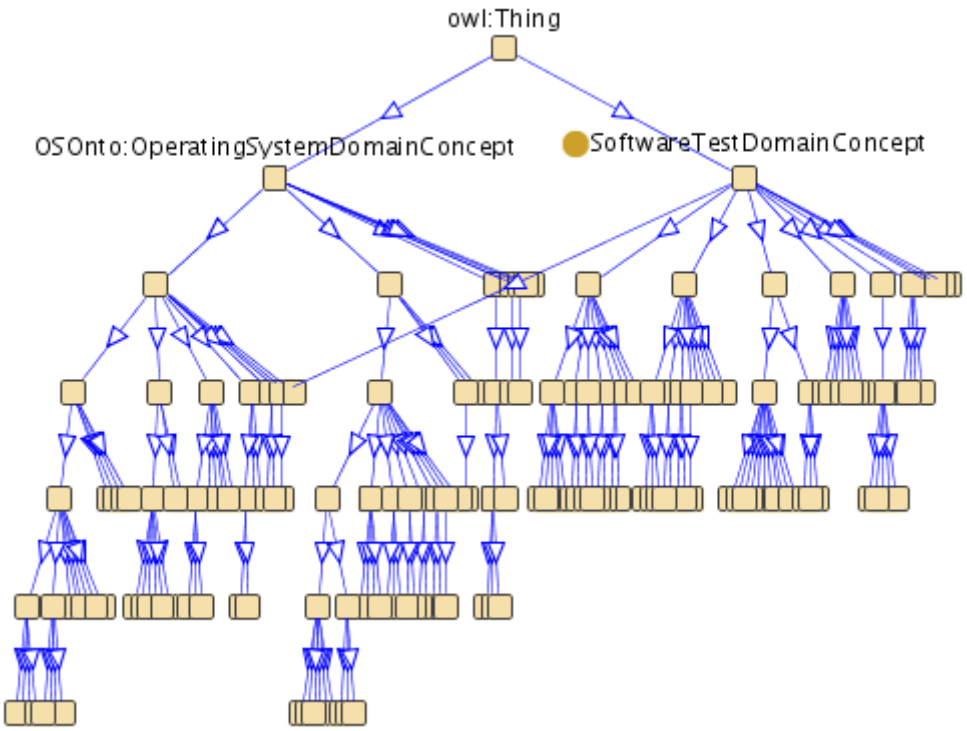

Figura 7.1: Árvore formada por classes e relações é-um da SwTO^I

a árvore de classes e relações é-um da *Pizza Ontology*. Por esta árvore é possível observar que esta ontologia também tem sete níveis desde o nó raiz até as folhas sendo que o quinto nível da raiz para as folhas é o mais denso, com maior quantidade de classes. Ao contrário da SwTO^I que tem o sétimo nível com bastante folhas, a *Pizza Ontology* apresenta uma única folha ou nó no sétimo nível. Este indicador contribui com a avaliação do tamanho da ontologia revelando as proporções da raíz as folhas e a profundidade da ontologia através da sua estrutura hierarquica. A tabela 7.4 mostra o total de classes por nível e destaca o quinto como o nível comum as duas ontologias que concentra a maior quantidade de classes.

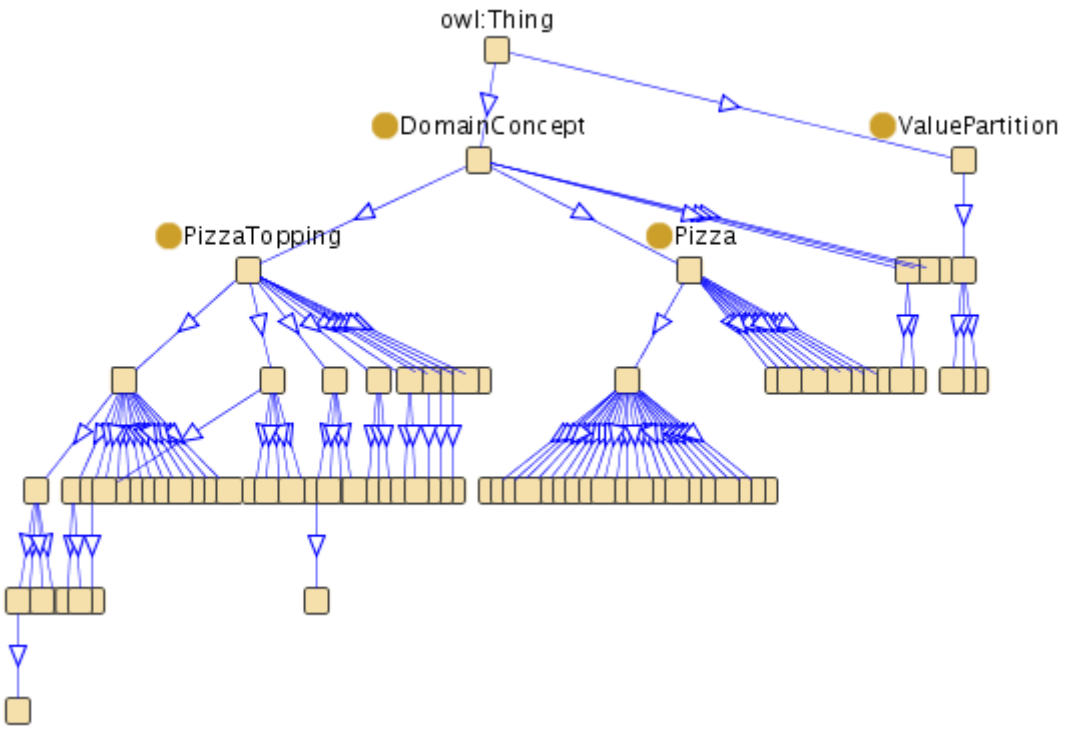

Figura 7.2: Árvore formada por classes e relações é-um da *Pizza Ontology*

Nível	SwTO^I	Pizza
1	1	1
2	2	2
3	17	6
4	48	26
5	**70**	**55**
6	38	8
7	18	1
Total de Classes	194	99

Tabela 7.4: Total de classes por níveis das ontologias

7.1.4 Classe de maior Grau da Ontologia quanto à Relação É-Um

Do mesmo grafo discutido na seção anterior é possível extrair a classe de maior grau da ontologia. Este indicador pode revelar as classes que possuem maior conexão (É-Um) com outras e contribuir com decisões de pojeto tanto para o aprimoramento da ontologia SwTO^I como para outras aplicações que tenham a intenção de reusá-

la. A tabela 7.5 informa a classe de maior grau É-Um por ontologia. Sendo o

Ontologia	Classe de Maior Grau é-um
SwTO^I	SoftwareTest(11)
Pizza	NamedPizza(23)

Tabela 7.5: Classe de maior grau *É-Um* por ontologia

domínio da SwTO^I o teste de software, é esperado que uma classe-chave para o domínio tenha o maior número de relações É-Um. Este indicador elicita a classe *SoftwareTest* (com grau igual a 11) como a de maior grau da SwTO^I em relação as demais classes. O mesmo pode ser observado na *Pizza Ontology*. O indicador destaca a classe *NamedPizza* (com grau igual a 23) como a classe de maior grau.

7.1.5 Classe de maior Grau da Ontologia quanto à Relação Todo-Parte

A relação Todo-Parte também estabelece um tipo de hierarquia entre os indivíduos de uma classe. Para exemplificar este tipo de relação, digamos que *carro* seja uma instância da classe *veículo* que por sua vez é composto pelas seguintes partes: motor, chassi, etc. Este tipo de relação nos permite enriquecer uma representação explicita de um domínio informando as partes de um todo. Para elaborar um grafo contendo a relação Todo-Parte é importante levar em consideração as características assumidas pelas propriedades como, por exemplo, propriedades inversas.

A Tabela 7.6 mostra a classe que contém maior quantidade de relações Todo-Parte por ontologia. Sendo o domínio da SwTO^I o teste de software, é esperado

Ontologia	Classe de Maior Grau Todo-Parte
SwTO^I	SoftwareTestProcess(20)
Pizza	Pizza(4)

Tabela 7.6: Classe de maior grau *Todo-Parte* por ontologia

que uma classe-chave para o domínio tenha o maior número de relações Todo-Parte.

Este indicador elicita a classe *SoftwareTestProcess* (com grau igual a 20) como a de maior grau da SwTO^I em relação as demais classes. O mesmo pode ser observado na *Pizza Ontology*. O indicador destaca a classe *Pizza* (com grau igual a 4) como a classe de maior grau.

A Tabela 7.7 sumariza os indicadores discutidos na Seção 7.1 e apresenta os resultados encontrados para a SwTO^I e *Pizza Ontology*.

Indicadores	SwTO^I	Pizza
Total de Classes Nomeadas	194	99
Média de P_O	0,73	0,08
Níveis (Relação É-Um)	7	7
Classe de maior grau (É-Um)	*SoftwareTest*(11)	*NamedPizza*(23)
Classe de maior grau (Todo-Parte)	*SoftwareTestProcess*(20)	*Pizza*(4)

Tabela 7.7: Resultados da avaliação quantitativa

7.1.6 Métricas apontadas pelo Protégé

Métricas extraídas da ontologia SwTO^I também podem ser observadas pelo editor Protégé. A seguir, a Figura 7.3 apresenta informações resumidas como: classes nomeadas, classes anônimas e propriedades. Se observarmos a Tabela 7.7 e a Figura 7.3 extraída do Protégé, veremos que existe uma diferença no total de classes. Isso ocorre porque o Protégé não adiciona em suas estatísticas a classe *owl:Thing* e não conta com a classe que está no nível que vem logo abaixo de *owl:Thing* porém, nesta avaliação, essas classes são levadas em consideração. As 192 classes apontadas pelo Protégé, mais *owl:Thing* e *TestDomainConcept* totalizam 194. As Figuras 7.4 e 7.5 apresenta, lado a lado, as métricas extraídas da SwTO e OSOnto respectivamente (da esquerda para a direita). Por estas figuras é possível observar que a OSOnto tem mais classes nomeadas que a SwTO mas, possuem a mesma quantidade de classes anônimas. Através do Protégé também foi possivel identificar a expressividade da ontologia SwTO^I. A Figura 7.6 mostra que a expressividade da SwTO^I foi identifi-

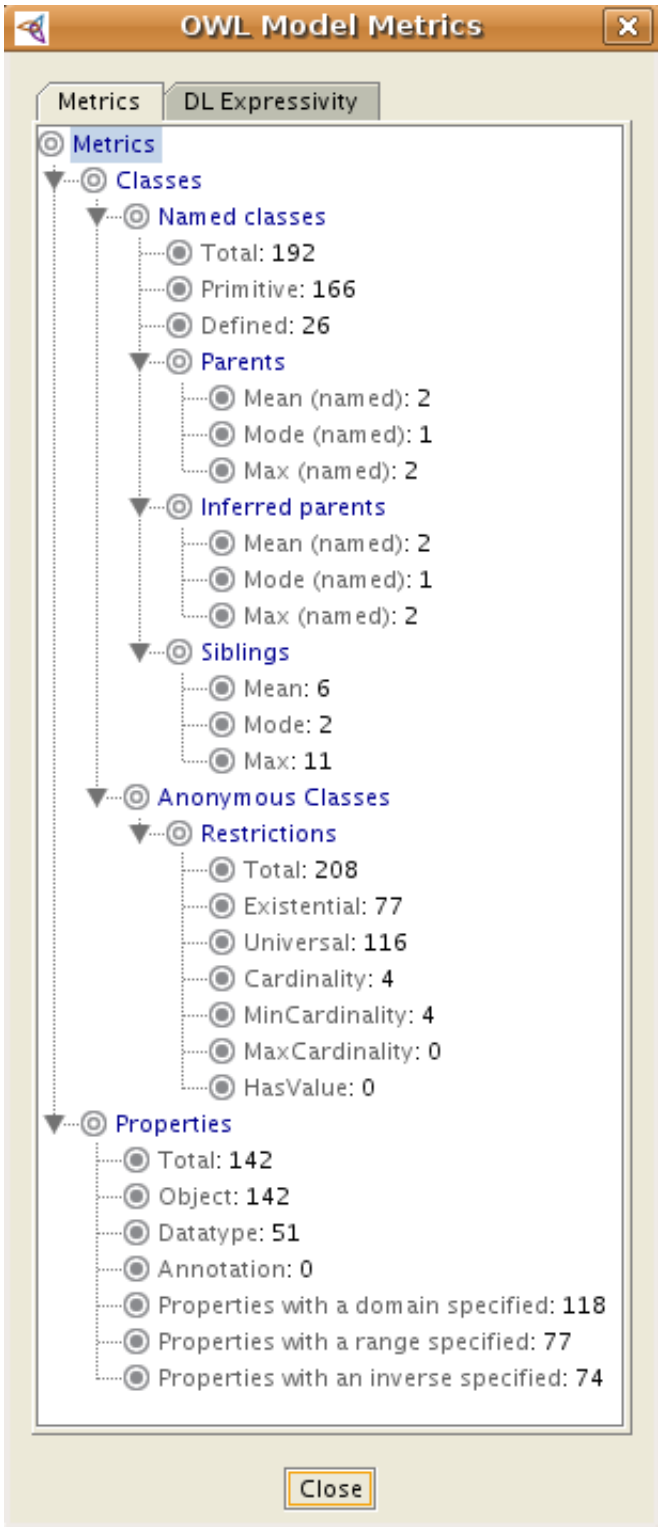

Figura 7.3: Métricas da SwTO^I

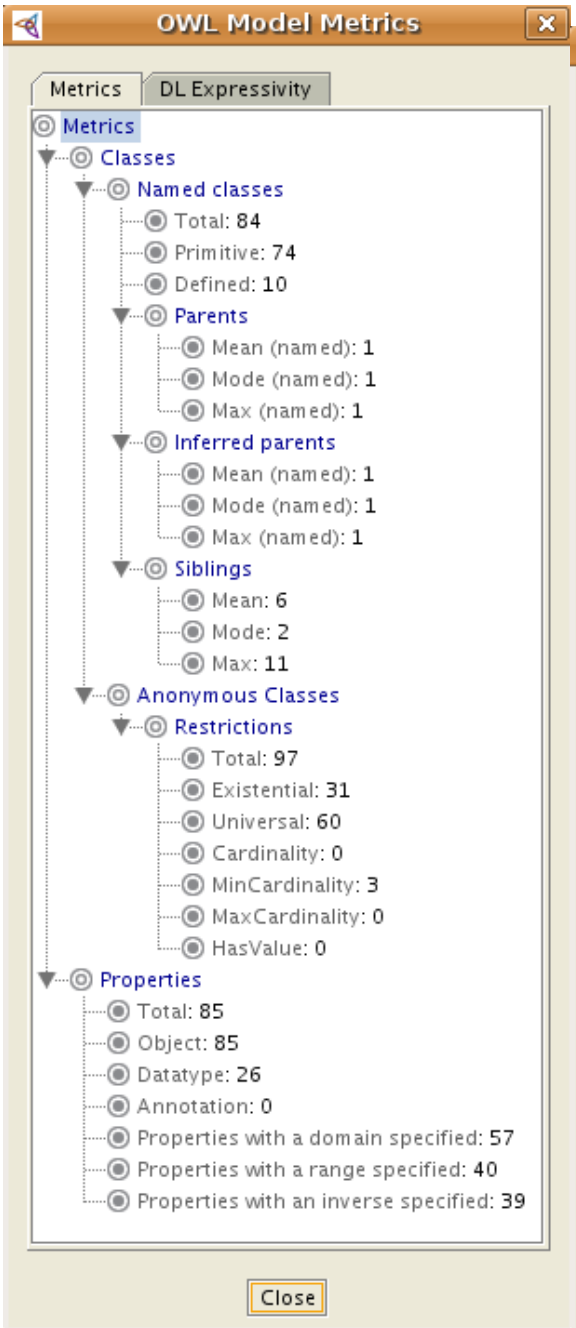

Figura 7.4: Métricas da SwTO

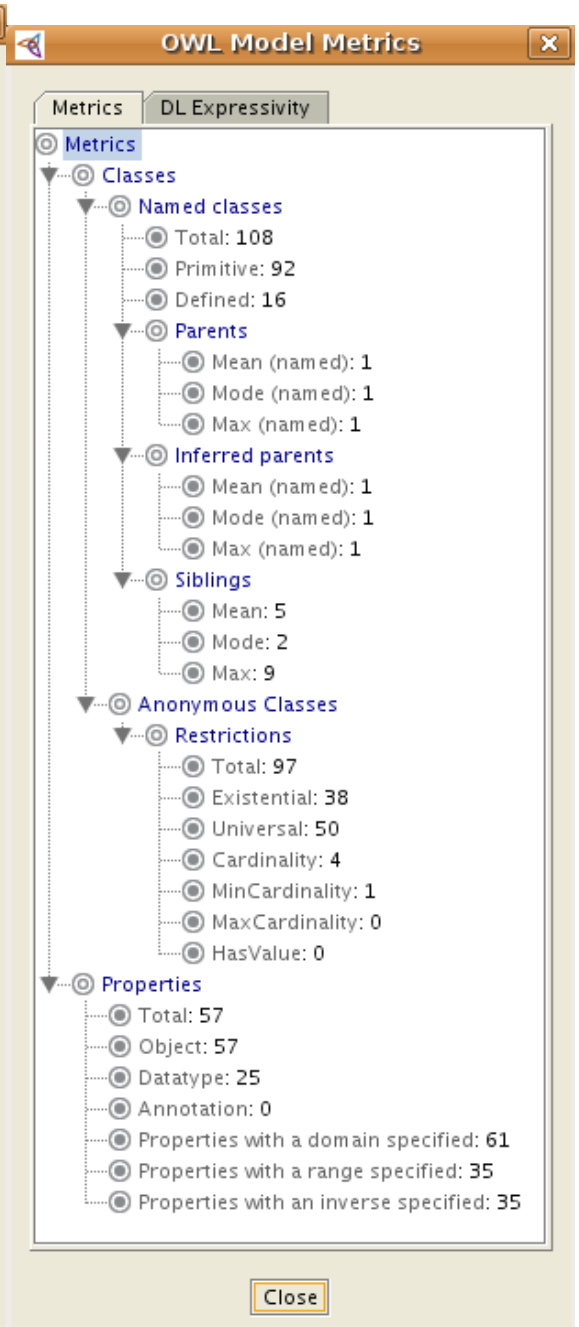

Figura 7.5: Métricas da OSOnto

cada como $\mathcal{SHOIQ}(D)$. Isto significa que a ontologia comporta regras transitivas, interseção entre classes, negação (complemento de uma classe), quantificação universal e existencial, e disjunção entre classes. Também há hierarquia de propriedades, restrição de valor, propriedades especificadas como inversas de outras, propriedades

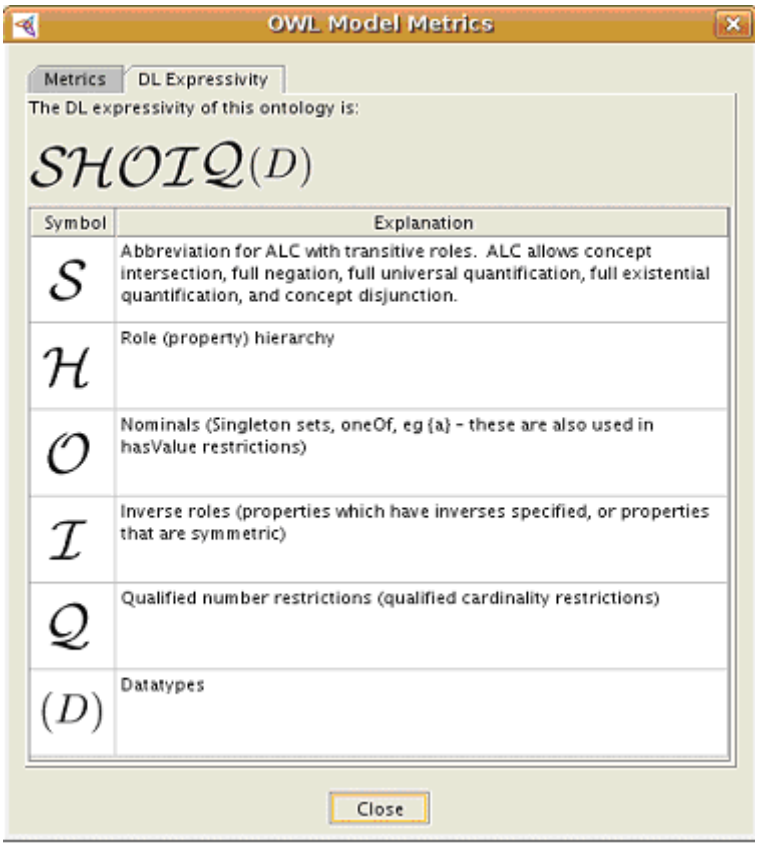

Figura 7.6: Expressividade das ontologias

simétricas, bem como restrições de cardinalidade.

A expressividade das ontologias OSOnto e SwTO também é $\mathcal{SHOIQ}$.

7.2 Avaliação Qualitativa

A avaliação qualitativa foi feita tendo em vista três critérios: consistência, completude e concisão.

No escopo de uma ontologia, uma dada definição é consistente se e somente se não há conhecimento contraditório inferido a partir de axiomas. O *critério de consistência* aponta resultados de inferências realizadas por um racionador sobre a teoria representada pela ontologia.

Se todos os conceitos de um determinado domínio forem representados por uma

ontologia e tal representação for comprovada por uma prova de completude, esta ontologia é dita completa. Sendo a ontologia uma especificação explícita de uma conceituação, que visa oferecer uma visão abstrata e simplificada do mundo que deseja-se representar para alguma finalidade, representar todos os conceitos pode tornar-se inviável para certos domínios. O *critério de completude* procura avaliar a representação explícita dos conceitos e apontar uma análise da completude de uma ontologia.

Uma ontologia é considerada concisa quando: (i) ela não armazena definições desnecessárias, (ii) não existe redundância entre definições dos termos e, (iii) redundâncias não são inferidas de outras definições. O critério de concisão avalia cada um desses pontos e aponta uma análise da concisão da ontologia.

Dos três critérios da avaliação qualitativa, o raciocinador Racer apoia o critério de consistência e concisão. Inferências foram realizadas na ontologia por intermédio da interface DIG (*Description Logic Implementers Group*), que funciona como um protocolo de comunicação, permitindo que raciocinadores interajam com o ambiente Protégé e conseqüentemente, com o código-fonte da ontologia. A seguir, os critérios serão detalhados.

7.2.1 Critério de Consistência

Os raciocinadores ajudam na verificação da consistência de três formas: (i) verificando se alguma condição definida resulta em conclusões contraditórias, (ii) inferindo a hierarquia de classes e subclasses e (iii) computando as instâncias inferidas. O Protégé, utilizado em conjunto com um raciocinador, também auxilia esta verificação mostrando a hierarquia definida pelo engenheiro da ontologia (em inglês, *asserted hierarchy*) e após a execução do raciocinador, o Protégé mostra a hierarquia inferida (em inglês, *inferred hierarchy*). Se alguma classe for reclassificada, ou seja, se sua superclasse mudar, ela será destacada em azul na hierarquia inferida. Se a

classe for inconsistente, será destacada em vermelho. A computação das instâncias inferidas, realizada pelo raciocinador, também pode ser visualizada pelo Protégé que mostra e totaliza as instâncias associadas a cada classe.

O raciocinador Racer concluiu que a ontologia SwTO^I é consistente já que nenhuma condição definida resultou em conclusões contraditórias. A Figura 7.7 mostra uma janela do editor Protégé contendo informações sobre a conclusão referente à consistência dos conceitos[1]. Esta mesma verificação de consistência foi realizada

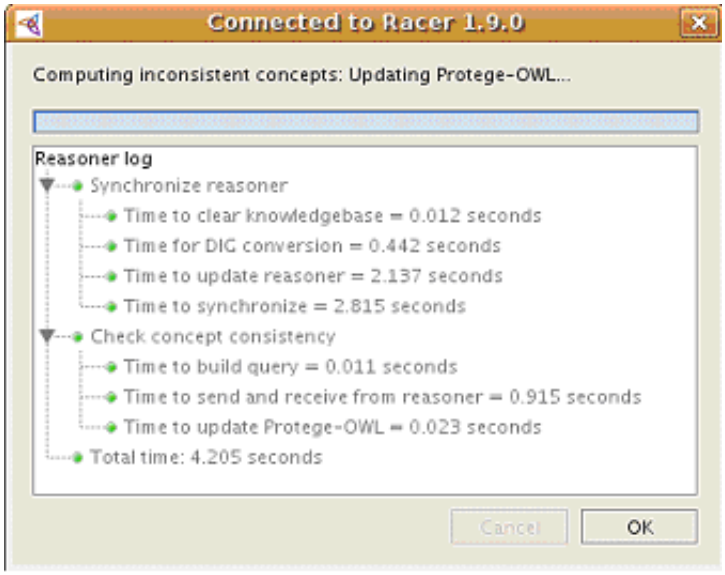

Figura 7.7: Resultados após a verificação de consistência da SwTO^I

para as ontologias OSOnto e SwTO. Ambas também são consistentes.

Em seguida foi realizada a verificação da taxonomia. A Figura 7.8 apresenta uma janela do editor Protégé informando sobre a verificação de consistência, hierarquia inferida e a computação das classes equivalentes. A partir da inferência realizada pelo Racer, o Protégé apresenta uma classificação diferente (destacada em azul pelo editor) porém esperada para as classes *Agent*, *BlackBox* e *WhiteBox*. A Figura 7.9 mostra a hierarquia inferida. A classe *Agent* é subclasse da classe união entre *OSOnto:Person* e *OSOnto:Software*. Isso fez com que a classe *Agent*

[1]Outra informações na Figura 7.7 referem-se à sincronização do raciocinador.

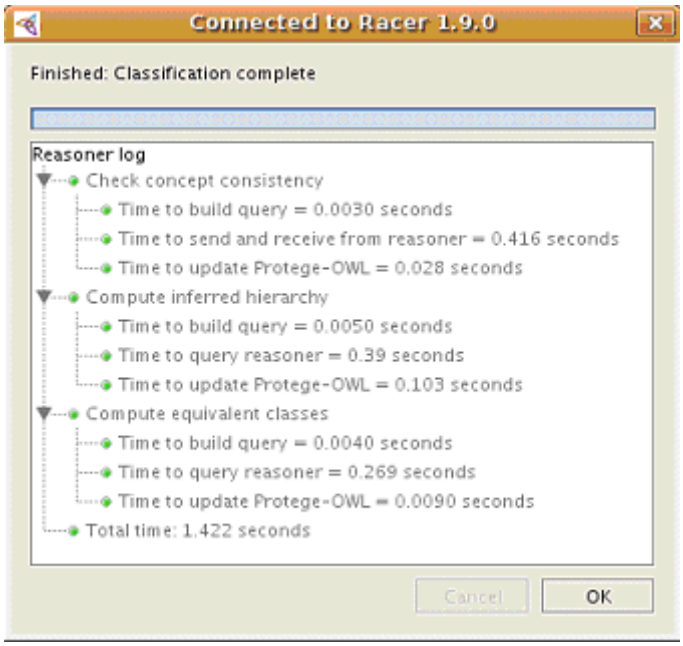

Figura 7.8: Resultados após a classificação de taxonomia da SwTO^I

fosse re-classificada como subclasse de *OSOnto:OperatingSystemDomainConcept*. A classe *GrayBox* é uma técnica de teste de *software* equivalente a união de outras duas técnicas de teste disjuntas entre si, a *BlackBox* e a *WhiteBox*. Na hierarquia definida essas três classes são irmãs, mas na hierarquia inferida, *BlackBox* e *WhiteBox* são subclasses de *GrayBox*. A classe *GrayBox* é equivalente a um axioma de cobertura que define que todas as suas instâncias pertencem também a uma de suas subclasses, ou seja, toda técnica *GrayBox* será exclusivamente *BlackBox* ou *WhiteBox*. A Figura 7.10 apresenta uma alerta do Protégé para a re-classificação inferida. A classe *TestTool* também aparece na taxonomia da OSOnto como subclasse de *OSOnto:Software* conforme a Figura 7.11). O Racer contribui também com a classificação das instâncias de cada classe. Todas as instâncias presentes na ontologia foram classificadas corretamente.

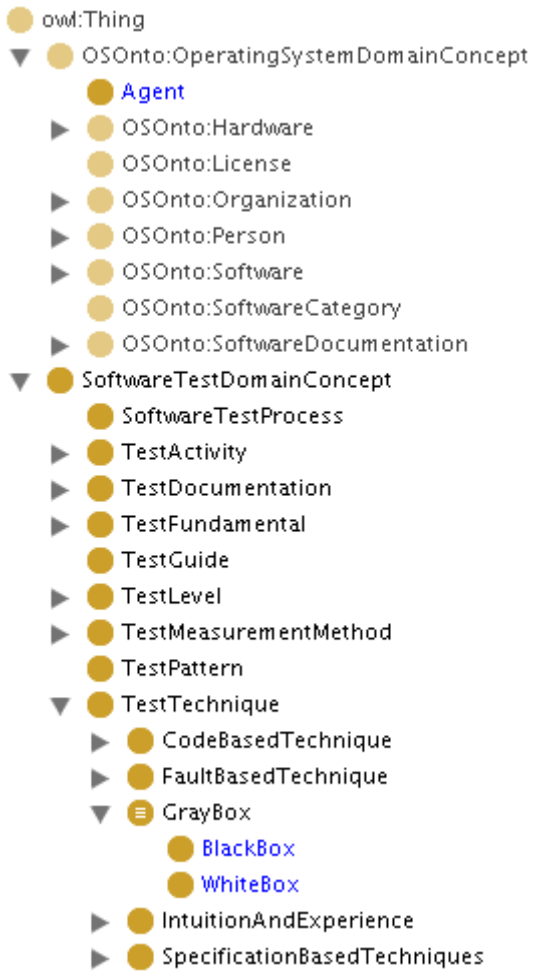

Figura 7.9: A hierarquia inferida da SwTO^I destacada pelo Protégé

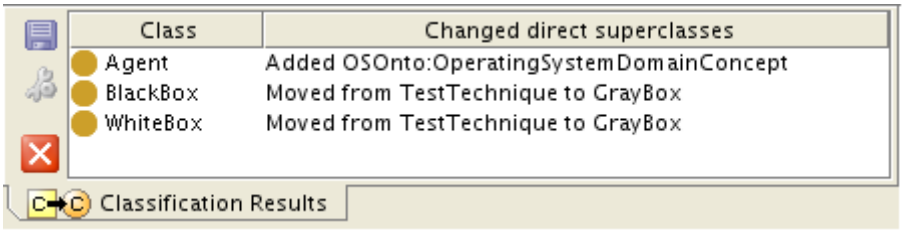

Figura 7.10: O resultado apontado pelo Protégé para a classificação da SwTO^I

7.2.2 Critério de Completude

As classes podem ser classificadas como primitivas ou definidas. Uma classe primitiva ou parcial é aquela que tem somente condições necessárias. Já as classes definidas ou completas são aquelas que tem pelo menos uma condição necessária

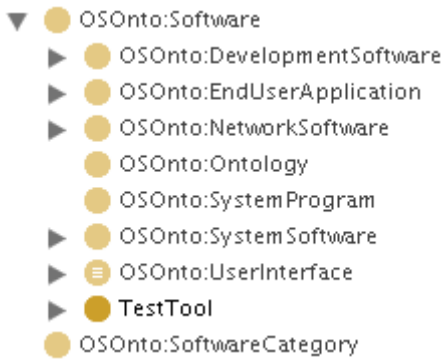

Figura 7.11: A classe *TestTool* na taxonomia inferida

e suficiente. Toda classe que não é definida é computada como primitiva já que toda classe terá pelo menos uma condição necessária (toda classe será subclasse de alguma classe, nem que seja de owl:Thing). Avaliando a quantidade de classes em uma ontologia que não possuem definições além da sua posição na hierarquia, é possível observar o nível de incompletude de cada classe e chegar a uma conclusão sobre o critério de completude da ontologia.

O segundo critério da avaliação qualitativa, *completude*, foi definido com base no total de classes primitivas, consideradas fracas em suas descrições. Das 194 classes nomeadas, 166 são primitivas. Isso corresponde a 85% das classes na ontologia. E dessas 194 classes, apenas 57 classes são realmente fracas na sua descrição, representando 30% do total de classes da SwTO^I. A Tabela 7.8 apresenta esta análise. De toda a ontologia SwTO^I, 30% das classes podem ser me-

Característica das Classes	Total	
Classes nomeadas	194	
Classes primitivas	166	85%
Classes primitivas com descrição fraca	57	**30%**

Tabela 7.8: Classes com descrição fraca

lhor representadas. São classes como *TestPattern*, *OSOnto:UserDocumentation*, *OSOnto:TechnicalDocumentation*, entre outras que não foram detalhadas devido à delimitação de escopo deste trabalho. Entretanto, essas classes são passíveis de

detalhamento em possíveis derivações deste trabalho.

7.2.3 Critério de Concisão

O critério de concisão da ontologia foi analisado de forma subjetiva, levando em consideração as revisões, observações destacadas pelos especialistas do grupo GIALix e a conclusão do raciocinador quanto a verificação de redundância entre definições dos termos e redundâncias que podem ser inferidas de outros axiomas.

, (ii) verificar se existe redundância entre definições dos termos e, (iii) verificar se redundâncias são inferidas de outras definições ou axiomas.

Para verificar se a ontologia armazenava definições desnecessárias os especialistas do grupo GIALix realizaram várias revisões e detectaram algumas definições como a da classe *Author*, que representava o conjunto de criadores de um *software*, *hardware* ou comunidade virutal. Segundo a avaliação dos especialistas, os criadores de um *hardware* ou de uma comunidade virtual estão fora do escopo da ontologia. Durante o desenvolvimento das ontologias SwTO^I, SwTO e OSOnto as definições desnecessárias apontadas pelos especialistas representaram um montante de 33% em relação a todas as definições das ontologias que foram devidamente removidas e na versão final apresentada por esta dissertação, estas definições desnecessárias não estão mais presentes.

As verificações de redundância entre definições dos termos e redundância inferida de outros axiomas foram realizadas com o auxílio do Racer que não detectou nenhum erro de circularidade já que este tipo de erro resultaria em uma inconsistência da ontologia e como foi dito anteriormente, a SwTO^I é consistente.

Os resultados do critério de concisão nos mostra que a ontologia SwTO^I é concisa e que durante o desenvolvimento a taxa de inconcisão chegou a 33%.

A tabela 7.9 apresenta um resumo da avaliação qualitativa contendo os resultados obtidos bem como seus respectivos valores em porcentagem.

Avaliação	Resultado
Consistente	100%
Incompleta	30%
Concisa	100%

Tabela 7.9: Resumo da avaliação qualitativa sobre a SwTO^I

7.3 Considerações Finais

A avaliação realizada na SwTO^I seguiu duas linhas: a avaliação quantitativa e a avaliação qualitativa.

A avaliação quantitativa se apoiou em métricas e indicadores que elicitam a estrutura da ontologia. A avaliação qualitativa se concentrou em basicamente três critérios: consistência, completude e concisão. Cada avaliação se apoiou em métodos diferentes e apontou qualidades e limitações da ontologia desenvolvida.

Um ponto positivo observado é que a ontologia é consistente, tem boa expressividade como conseqüência da sua estrutura bem definida formalmente, com 70% das classes ricas em suas descrições e foi efetivamente utilizada pela ferramenta TeSG não só para gerar testes para o Linux como para o SiGIPós, comprovando que a SwTO^I atende não só a um domínio específico de conhecimento mas também à solução de geração de seqüências de teste.

Capítulo 8

Discussão e Conclusões

Este trabalho propõe uma família de ontologias composta por OSOnto, SwTO e SwTO^I. A SwTO^I oferecer um recurso de representação de conhecimento que, como demonstrado pela TeSG, pode ser aplicado em sistemas que lidam com o teste do Linux, aplicação esta que pode se estender a outros sistemas inseridos em paradigmas diferentes de *software* livre.

A Engenharia de Software procura consolidar métodos que otimizem o desenvolvimento de sistemas de informação de boa qualidade e com o menor custo e esforço possíveis. O paradigma de *software* livre é um dos que contribui para otimização de processo de desenvolvimento de *software* através de seus exemplos de projetos bem sucedidos. O projeto do Linux, mesmo sendo considerado de referência, também se depara com necessidades e incógnitas quanto a sua qualidade e processo de desenvolvimento. A observação da comunidade do LTP (*Linux Test Project*) e suas necessidades de uma melhor organização do conhecimento referente ao teste do Linux nos motivou a desenvolver este trabalho de pesquisa.

A formalização semântica de conhecimento foi escolhida como caminho alternativo aos já explorados caminho pela comunidade do LTP no intuito de contribuir com a redução de ambigüidade nos atefatos de teste (casos de teste, *scripts*, documentação, etc.), com um melhor compartilhamento do conhecimento entre os projetistas quanto a critérios de geração, seleção e automação de teste, bem como com

um vocabulário que pode facilitar em algumas situações o consenso entre membros da comunidade.

O desenvolvimento da ontologia SwTO^I passou por quatro fases: (i) a identificação do objetivo e escopo, (ii) a construção, (iii) a avaliação e (iv) a documentação.

A primeira fase foi marcada pela complexidade inerente ao tema abordado e dificultada pela ausência de um quadro teórico completo compreendendo todos os aspectos envolvidos no processo de teste do Linux. A segunda fase exigiu um aprendizado das técnicas, ferramentas e linguagem utilizadas na construção das ontologias. A terceira fase foi marcada por dificuldades atribuídas à inexistência de um referencial teórico consolidado para a avaliação de ontologias. Os métodos e técnicas documentados são vagos e faltam ferramentas que auxiliem o processo. A quarta fase foi marcada pela documentação da ontologia e elaboração do OWL DOC.

A pesquisa foi desenvolvida tendo como um objetivo a aplicabilidade das ontologias, isto é, que pudessem ser utilizadas. A aplicabilidade foi demonstrada atavés da ferramenta TeSG.

A ontologia SwTO^I em sua representação de conhecimento relativo ao teste do Linux envolve uma parcela significativa de conceitos referentes tanto ao domínio de sistemas operacionais quanto ao de teste de *software*. A ontologia não é completa porém consistente e concisa.

8.1 Limitações

Em atividades de representação do conhecimento, a interação e cooperação entre vários indivíduos é desejável. O conhecimento é dinâmico, está em constante evolução, um dos desafios da representação de conhecimento.

A engenharia das ontologias foi praticamente realizada por um único indivíduo, inserido em um grupo de pesquisa formado por especialistas do domínio. No decorrer do desenvolvimento da ontologia, esses especialistas participaram com a revisão de

resultados parciais do projeto. Alguns eram especialistas em teste do Linux, outros em ontologias. Através deles, com o avanço do projeto, ocorreu interação entre as duas áreas de conhecimento porém no escopo de um pequeno grupo de pesquisa.

Até o presente momento, a cooperação ainda não ocorreu envolvendo especialistas externos ao grupo de pesquisa. Uma forma de viabilizar esta cooperação na web as ontologias em um ambiente que possibilitasse a participação voluntária d eoutros desenvolvedores na engenharia ou instanciação das ontologias.

Existem melhorias a serem feitas na ontologia SwTO^I. Conceitos podem ser melhor representados e/ou adicionados.

8.2 Trabalho Futuro

No decorrer deste trabalho surgiram alguns pontos que não foram tratados devido à delimitação do escopo inicialmente proposto. Estes pontos de pesquisa, que envolvem os principais temas de continuação considerados para este trabalho são os seguintes:

- Enriquecer a OSOnto com mais conhecimento sobre o Linux.
- Estender a ontologia OSOnto, inserindo conhecimento sobre outro *kernel*;
- Enriquecer a SwTO^I com mais conhecimento sobre teste de software, detalhando melhor algumas classes como níveis de teste, padrões, métodos de mensuração, técnicas de teste e ferramentas.
- Enriquecer a SwTO^I com mais conhecimento sobre o teste do Linux.
- Enriquecer a SwTO^I com mais conhecimento sobre outros tipos de *software*.
- Avaliar a ontologia através de outros métodos.

Referências Bibliográficas

[1] A. Abran, J. Moore, P. Bourque, R. Dupuis, and L. Tripp. *Guide to the Software Engineering Body of Knowledge - SWEBOK.* IEEE Press, Piscataway, NJ, USA, 2004.

[2] M. Afsharchi, B. Far, and J. Denzinger. Learning non-unanimous ontology concepts to communicate with groups of agents. In *IAT '06: Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology*, pages 211–217, Washington, DC, USA, 2006. IEEE Computer Society.

[3] F. Baader. *The description logic handbook: theory, implementation, and application.* United Kingdom: Cambridge University Press, 2003.

[4] A. Bauer and M. Pizka. The contribution of free software to software evolution. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 170, Washington, DC, USA, 2003. IEEE Computer Society.

[5] R. Brachman and H. Levesque. *Knowledge Representation and Reasoning.* San Francisco: Elsevier, 2004.

[6] A. Costa. *Sistema Gestor de Informação do Programa de Pós-graduação em Informática.* Monografia de conclusão do curso de Bacharelado em Ciência da Computação - UFAM., 2004.

[7] A. Costa. Geração de seqüências de teste com o auxílio de ontologias. Master's thesis, Universidade Federal do Amazonas, 2007.

[8] K. Crowston, H. Annabi, J. Howison, and C. Masango. Effective work practices for software engineering: free/libre open source software development. In *WISER '04: Proceedings of the 2004 ACM Workshop on Interdisciplinary Software Engineering Research*, pages 18–26, New York, NY, USA, 2004. ACM Press.

[9] R. Denaux. Ontology-based interactive user modeling for adaptative web information systems. Master's thesis, Technische Universiteit Eindhoven, 2005.

[10] Portal do Linux. Disponível em: http://www.kernel.org. Acessado em: 11-Fevereiro-2008.

[11] Debian's Quality Assurance Group. Disponível em: http://qa.debian.org/. Acessado em: 11-Fevereiro-2008.

[12] Gnome's Quality Assurance Group. Disponível em: http://developer.gnome.org/projects/bugsquad/. Acessado em: 11-Fevereiro-2008.

[13] N. Hinds. Kernel korner: the linux test project. *Linux Journal*, 2005(129):12, 2005.

[14] M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C. Wroe. A practical guide to building owl ontologies using the protégé-owl plugin and co-0de tools edition 1.0. Technical report, University of Manchester, 2004.

[15] N. Huang and S. Diao. Structure-based ontology evaluation. In *IAT '06: Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology*, pages 211–217, Washington, DC, USA, 2006. IEEE Computer Society.

[16] M. Jae and S. Lee. Essence of distributed work: The case of the linux kernel. *Journal of First-Monday Open Source*, 5, November 2000.

[17] Y. Kishida and K. Ye. Toward an understanding of the motivation open source software developers. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 419–429, Washington, DC, USA, 2003. IEEE Computer Society.

[18] C. Larman. *Utilizando UML e Padrões: Uma introdução à análise e ao Projeto Orientado a Objeto*. Porto Alegre: Bookman, 2000.

[19] C. Lattemann and S. Stieglitz. Framework for governance in open source communities. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 7*, page 192, Washington, DC, USA, 2005. IEEE Computer Society.

[20] R. Love. *Linux Kernel Development. 2.ed.* Novell Press, 2005.

[21] Linux Test Project (LTP). Disponível em: http://www.linuxtestproject.org/. Acessado em: 11-Fevereiro-2008.

[22] M. Michlmayr. Managing volunteer activity in free software projects. In *Proceedings of the 2004 USENIX Annual Technical Conference, FREENIX Track*, pages 93–102, Boston, USA, 2004.

[23] Basic Linux Ontology. Disponível em: http://wwwis.win.tue.nl:8080/šwale/blo. Acessado em: 11-Fevereiro-2008.

[24] Pizza Ontology. Disponível em: http://www.co-ode.org/ontologies/pizza. Acessado em: 11-Fevereiro-2008.

[25] Tutorial OWL. Disponível em: http://www.w3.org/TR/owl-guide. Acessado em: 11-Fevereiro-2008.

[26] Raciocinador Pellet. Disponível em: http://pellet.owldl.com. Acessado em: 11-Fevereiro-2008.

[27] R. Pressman. *Engenharia de Software (5.ed.)*. Rio de Janeiro: McGraw-Hill, 2002.

[28] Editor Protégé. Disponível em: http://protege.stanford.edu. Acessado em: 11-Fevereiro-2008.

[29] Raciocinador Racer. Disponível em: http://www.racer-system.com. Acessado em: 11-Fevereiro-2008.

[30] U. Raja and E. Barry. Investigating quality in large-scale open source software. In *5-WOSSE: Proceedings of the Fifth Workshop on Open Source Software Engineering*, pages 1–4, New York, NY, USA, 2005. ACM Press.

[31] SwTO e OSOnto. Repositório SVN do projeto TestLix com o código-fonte e documentação da SwTO^I. Disponível em: http://moemais.ufam.edu.br/svn/GIALix/trunk/testlix. Acessado em: 11-Fevereiro-2008.

[32] J. Sowa. *Knowledge Representation: logical, philosophical, and computational foundations*. Pacific Grouve: Books/Cole, 2000.

[33] S. Staab and R. Studer. *Handbook on Ontologies*. Berlin: Springer-Verlag, 2004.

[34] Projeto Swebok. Disponível em: http://www.swebok.org. Acessado em: 11-Fevereiro-2008.

[35] J. Tanomaru. Motivação, fundamentos e aplicações de algoritmos genéticos. Technical report, 1995.

[36] KDE Quality Team. Disponível em: http://quality.kde.org/. Acessado em: 11-Fevereiro-2008.

[37] M. Uschold and M. Grüninger. Ontologies: principles, methods, and applications. *Jornal of Knowledge Engineering Review*, 11(2):93–155, 1996.

[38] L. Zhao and S. Elbaum. A survey on quality related activities in open source. *Jornal of SIGSOFT Softw. Eng. Notes*, 25(3):54–57, 2000.

[39] L. Zhao and S. Elbaum. Quality assurance under the open source development model. *Jornal of System Software*, 66(1):65–75, 2003.

[40] N. Ziviani. *Projeto de Algoritmos. 2.ed.* Thomson Pioneira, 2004.

Apêndice A

OSOnto: uma ontologia de Sistemas Operacionais

O objetivo deste Apêndice é apresentar a OSOnto *(Operating Systems Ontology)* [31] que foi construída com base no método de Uschold e Gruninger [37]. Durante a primeira fase do desenvolvimento da OSOnto, identificação do objetivo e escopo, foram definidas questões de competência em linguagem natural descritas na Seção A.1. A segunda fase, construção da ontologia, foi marcada pela pesquisa de outras possíveis ontologias que tratassem do mesmo domínio, bem como a avaliação de um possível reuso do conhecimento, a codificação em OWL DL, uso do Racer e Pellet como raciocinadores e do editor Protégé. A BLO (*Basic Linux Ontology*) foi apontada pela pesquisa realizada na segunda fase como a ontologia mais relevante. A justificativa para a escolha da BLO e posterior reuso do conhecimento deveu-se ao fato desta ontologia representar conceitos associados ao Linux. Sendo o Linux um *kernel*, parte integrante de um sistema operacional e foco de investigação deste trabalho, a BLO se mostrou adequada aos objetivos definidos no Capítulo 1. A avaliação individual da OSOnto, terceira fase prevista no método, não faz parte do escopo deste trabalho e sim a avaliação da SwTO^{I} que contém a OSOnto. A atividade proposta na quarta fase do método, a documentação, será discutida na Seção A.2. A quinta e última fase do método adotado, definição de um guia para a

ontologia, foi feita através da geração do OWL DOC e está disponível no repositório do projeto.

A.1 Questões de competência da OSOnto

O escopo da OSOnto foi definido por meio de questões de competência que descrevem em linguagem natural, e de forma interrotativa, os requisitos que esta deve atender.

Essas questões de competência definem o escopo da ontologia em relação à sua capacidade de representar os conceitos do domínio de sistemas operacionais.

Cabe ressaltar que sendo o sistema operacional um software, foi necessário adicionar conceitos de mais alto nível, associados à sistemas operacionais, com o objetivo de definí-los melhor e disponibilizar a ontologia de forma que ela possa ser reusada e extendida mais facilmente. A tabela A.1 apresenta algumas questões de competência da OSOnto que guiaram a definição do escopo.

A.2 A OSOnto

A seguir o conhecimento formal representado pela OSOnto será apresentado. Para facilitar a exposição e a compreensão do conhecimento optou-se por particionar a ontologia em suas classes nomeadas como nível de granularidade. A numeração adotada nas seções que destacam as classes tem relação com a taxonomia. A partir das classes nomeadas serão detalhados todos os outros conceitos relacionados como propriedades, instâncias e condições lógicas associadas. Os formalismos adotados para representar os conceitos da OSOnto na Seção seguinte são: lógica de descrição, fragmentos do código OWL DL e diagramas.

- ***OperatingSystemDomainConcept:*** são subclasses desta classe, todos os conceitos que fazem parte ou que têm alguma relação com o domínio de sistemas operacionais.
  OperatingSystemDomainConcept *é a classe de mais alto nível na ontologia e é subclasse*

Conceito	Questões à serem respondidas
Hardware	Este tipo de *hardware* interage com que tipo de sistema operacional?
Licença	Esta licença está associada com que categoria de software? Qual a licença de um determinado sistema operacional? Quem criou esta licença?
Organização	Quem são as pessoas que fazem parte desta organização? O que esta organização produz (licença, software, *hardware* ou documentação)?
Pessoa	Que papeis as pessoas podem assumir? Quem são os usuários? Aprovadores? Desenvolvedores? Autores? Mantenedores? Revisores? Quem são as pessoas que exercem mais de um papel? Quem são as pessoas que estão envolvidas com o processo de um sistema operacional?
Sistema Operacional	Quais são as partes de um sistema operacional? Quais os subsistemas de um kernel? Quais as possíveis operações sobre objetos do VFS do Linux?
Categorias de Software	Quais são as categorias de software? Quais são as licenças associadas a essas categorias?
Documentação	Quais são as documentações de um determinado software?
Comunidades Virtuais	Quais são as comunidades virtuais que tem interesse por um determinado sistema operacional?

Tabela A.1: Questões de competência da OSOnto

*de **owl:Thing**. A classe **owl:Thing** representa um conjunto que contém todos os indivíduos possíveis. Devido a este princípio, qualquer outra classe será subclasse de **owl:Thing**.*

$$\texttt{OperatingSystemDomainConcept} \sqsubseteq \top$$

A figura A.1 introduz as subclasses de *OperatingSystemDomainConcept*.

- ***Hardware:*** representa o conjunto dos componentes físicos compostos por partes mecânicas e/ou eletrônicas. Freqüentemente o *hardware* interage com um tipo especial de *software* (em inglês, *System Software*).
  Condições Necessárias:

  (i) ***Hardware*** é subclasse de ***OperatingSystemDomainConcept*** por ser um conceito que faz parte do domínio de Sistemas Operacionais.

  $$\texttt{Hardware} \sqsubseteq \texttt{OperatingSystemDomainConcept}$$

  (ii) Indivíduos da classe ***Hardware*** só relacionam-se através da propriedade ***interactsWith***

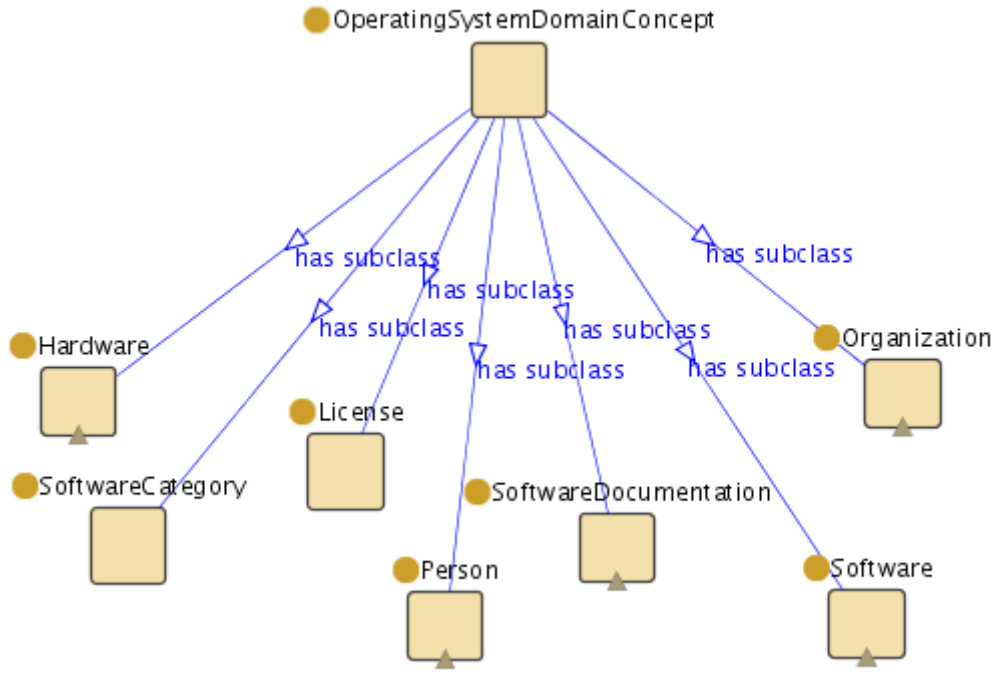

Figura A.1: A classe *OperatingSystemDomainConcept* e suas subclasses

com indivíduos da classe ***SystemSoftware***.

$$\texttt{Hardware} \sqsubseteq \forall\ \texttt{interactsWith}\ .\ \texttt{SystemSoftware}$$

(iii) Indivíduos da classe ***Hardware*** relacionam-se através da propriedade ***interactsWith***) com pelo menos um indivíduo da classe ***SystemSoftware***.

$$\texttt{Hardware} \sqsubseteq \exists\ \texttt{interactsWith}\ .\ \texttt{SystemSoftware}$$

(iv) Indivíduos da classe ***Hardware*** relacionam-se através da propriedade ***isComposedOfAnother***) com pelo menos um indivíduo da mesma classe.

$$\texttt{Hardware} \sqsubseteq \exists\ \texttt{isComposedOfAnother}\ .\ \texttt{Hardware}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***Hardware****, pode-se dizer:*

(i) ***interactsWith*** é uma propriedade objeto.

$$\texttt{interactsWith} \in \texttt{P}_\texttt{O}$$

(ii) ***isComposedOfAnother*** é uma propriedade objeto transitiva (Equações A.1 e A.2). Para exemplificar a transitividade da propriedade na classe anônima $\exists$*isComposedOfAnother* . *Hardware*, digamos que se uma impressora é composta por uma placa controladora, que por sua vez é composta por componentes eletrônicos (tais como capacitores, diodos e resistores), por transitividade, é possível inferir que uma impressora é composta

por componentes eletrônicos.

$$\texttt{isComposedOfAnother} \in \texttt{P}_0 \tag{A.1}$$

$$\texttt{isComposedOfAnother} \in \mathbf{R}_+ \tag{A.2}$$

(iii) Geralmente o *hardware* dispõe de um código para representar o seu modelo. A propriedade ***hasHardwareModel*** representa este código que costuma ser importante para a identificação de um *hardware*.
hasHardwareModel é uma propriedade de tipo de dados funcional (Equações A.3 e A.4). Para esta propriedade, o domínio especificado é a classe ***Hardware*** (Equação A.5) e o contradomínio é **XMLSchema:string** (Equação A.6).

$$\texttt{hasHardwareModel} \in \texttt{P}_\texttt{D} \tag{A.3}$$

$$\top \sqsubseteq (\leq \texttt{1 hasHardwareModel}) \tag{A.4}$$

$$\top \sqsubseteq \forall \texttt{hasHardwareModel} \,.\, \texttt{Hardware}\ (\texttt{Hardware} \neq \bot) \tag{A.5}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasHardwareModel}^- \,.\, \texttt{XMLSchema:string} \\ (\texttt{XMLSchema:string} \neq \bot) \end{pmatrix} \tag{A.6}$$

(iv) Freqüentemente o *hardware* dispõe de um código para representar o seu número de série. A propriedade ***hasHardwareSerialNumber*** representa este código que costuma ser importante para a identificação de um *hardware*. Esta propriedade é de tipo de dados (Equação A.7), funcional (Equação A.8), cujo domínio é a classe ***Hardware*** (Equação A.9) e o contradomínio é **XMLSchema:string** (Equação A.10).

$$\texttt{hasHardwareSerialNumber} \in \texttt{P}_\texttt{D} \tag{A.7}$$

$$\top \sqsubseteq (\leq \texttt{1 hasHardwareSerialNumber}) \tag{A.8}$$

$$\top \sqsubseteq \forall \texttt{hasHardwareSerialNumber} \,.\, \texttt{Hardware}\ (\texttt{Hardware} \neq \bot) \tag{A.9}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasHardwareSerialNumber}^- \,.\, \texttt{XMLSchema:string} \\ (\texttt{XMLSchema:string} \neq \bot) \end{pmatrix} \tag{A.10}$$

(v) A propriedade ***hasManufacturer*** representa o fabricante que produz o *hardware*. Esta é uma propriedade de tipo de dados (Equação A.11), funcional (Equação A.12) e tem como domínio a classe ***Hardware*** (Equação A.13) e o contradomínio é **XMLSchema:string** (Equação A.14). Para esta propriedade foram adicionados alguns valores possíveis. A Equação A.15 apresenta esses valores.

$$\texttt{hasManufacturer} \in \texttt{P}_\texttt{D} \tag{A.11}$$

$$\top \sqsubseteq (\leq \texttt{1 hasManufacturer}) \tag{A.12}$$

$$\top \sqsubseteq \forall \texttt{hasManufacturer} \,.\, \texttt{Hardware}\ (\texttt{Hardware} \neq \bot) \tag{A.13}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasManufacturer}^- \,.\, \texttt{XMLSchema:string} \\ (\texttt{XMLSchema:string} \neq \bot) \end{pmatrix} \tag{A.14}$$

$$\{ \begin{pmatrix} \texttt{IBM}, \texttt{Dell}, \texttt{Compaq}, \texttt{Sony}, \texttt{Toshiba}, \\ \texttt{Apple}, \texttt{Intell}, \texttt{LG}, \texttt{Seagate}, \texttt{Motorola} \end{pmatrix} \} \sqsubseteq \texttt{hasManufacturer} \tag{A.15}$$

O fragmento do código fonte A.2 da OSOnto mostra, da linha 5 a 9 o construtor owl:oneOf e a primeira instância definida à propriedade ***hasManufacturer***.

Disjunções: A classe ***Hardware*** é disjunta das classes ***Person***, ***Software***, ***License***, ***Organization*** e ***SoftwareCategory*** já que estas classes não compartilham instâncias.

$$\texttt{Hardware} \sqsubseteq \begin{pmatrix} \neg\, \texttt{Person} \sqcap \neg\, \texttt{Software} \sqcap \neg\, \texttt{License} \\ \sqcap \neg\, \texttt{Organization} \sqcap \neg\, \texttt{SoftwareCategory} \end{pmatrix}$$

```
</owl:FunctionalProperty>
  <owl:FunctionalProperty rdf:ID="manufacturer">
    <rdfs:range>
      <owl:DataRange>
        <owl:oneOf rdf:parseType="Resource">
          <rdf:rest rdf:parseType="Resource">
            <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >IBM</rdf:first>
                </owl:oneOf>
            ...
      </owl:DataRange>
    </rdfs:range>
    <rdfs:domain rdf:resource="#Hardware"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>
```

Código A.1: *Construtor oneOf em OWL*

Instanciação: O exemplo a seguir demonstra instâncias da classe ***Hardware***.

$$\{\texttt{Hw22SIH34}, \texttt{Hw3HS0XJ94}, \texttt{Hw6952WSX}, \texttt{HwKOD684WS}\} \sqsubseteq \texttt{Hardware}$$

A seguir é possível observar a relação de uma das instâncias com as propriedades P_D que descrevem a classe ***Hardware***.

$$\texttt{hasHardwareModel}(\texttt{Hw22SIH34}, \texttt{KU8E832422})$$
$$\texttt{hasHardwareSerialNumber}(\texttt{Hw22SIH34}, \texttt{22SIH34})$$
$$\texttt{hasManufacturer}(\texttt{Hw22SIH34}, \texttt{LG})$$

A instância **Hw22SIH34** também possui as propriedades ***rdfs:comment*** e ***rdfs:label*** que complementam a descrição desta instância conforme a seguir:

$$\texttt{rdfs} : \texttt{comment}(\texttt{Hw22SIH34}, \text{Super Multi DVD Driver})$$
$$\texttt{rdfs} : \texttt{label}(\texttt{Hw22SIH34}, \text{DVD Driver})$$

De forma análoga, as outras instâncias da classe ***Hardware*** também são descritas pelas mesmas propriedades conforme a seguir:

$$\texttt{rdfs} : \texttt{comment}(\texttt{Hw3HS0XJ94}, \text{Hard Disk Barracuda ATA IV})$$
$$\texttt{rdfs} : \texttt{label}(\texttt{Hw3HS0XJ94}, \text{Hard Disk})$$
$$\texttt{rdfs} : \texttt{comment}(\texttt{Hw6952WSX}, \text{Fax Modem 56Kbps Data Download Rates})$$
$$\texttt{rdfs} : \texttt{label}(\texttt{Hw6952WSX}, \text{Fax Modem ENF656})$$
$$\texttt{rdfs} : \texttt{comment}(\texttt{HwKOD684WS}, \text{Pentium IV})$$
$$\texttt{rdfs} : \texttt{label}(\texttt{HwKOD684WS}, \text{Processor Intel})$$

- ***ComputingDevice:*** representa um tipo de *hardware* capaz de realizar computação em um determinado tempo. Em geral, um recurso computacional é composto por um processador.
  Condições Necessárias:

(i) ***ComputingDevice*** é subclasse de ***Hardware*** por ser um tipo de *hardware* capaz de realizar um processamento.

$$\texttt{ComputingDevice} \sqsubseteq \texttt{Hardware}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe ***ComputingDevice***.

$$\{\ \texttt{Cd\_4IEJ883IK}\ \} \sqsubseteq \texttt{ComputingDevice}$$

A seguir é possível observar a relação de uma das instâncias da classe ***ComputingDevice*** com as propriedades P_O e P_D que foram herdadas da classe pai.

$$\begin{aligned}
\texttt{hasHardwareModel}(\texttt{Cd_4IEJ883IK}, \texttt{IBM87676})\\
\texttt{hasHardwareSerialNumber}(\texttt{Cd_4IEJ883IK}, \texttt{A876C8E3})\\
\texttt{hasManufacturer}(\texttt{Cd_4IEJ883IK}, \texttt{IBM})\\
\texttt{interactsWith}(\texttt{Cd_4IEJ883IK}, \texttt{Ubuntu})\\
\texttt{isComposedOf}(\texttt{Cd_4IEJ883IK}, \texttt{Hw22SIH34})\\
\texttt{isComposedOf}(\texttt{Cd_4IEJ883IK}, \texttt{Hw3HS0XJ94})\\
\texttt{isComposedOf}(\texttt{Cd_4IEJ883IK}, \texttt{Hw6952WSX})\\
\texttt{isComposedOf}(\texttt{Cd_4IEJ883IK}, \texttt{HwKOD684WS})
\end{aligned}$$

Para complementar a descrição da instância **Cd_4IEJ883IK** temos as propriedades ***rdfs:comment*** e ***rdfs:label***:

$$\begin{aligned}
\texttt{rdfs} : \texttt{comment}(\texttt{Cd_4IEJ883IK}, \text{A personal computer NetVista})\\
\texttt{rdfs} : \texttt{label}(\texttt{Hw3HS0XJ94}, \text{PCNetVista})
\end{aligned}$$

- ***License:*** esta classe representa os contratos ou documentos que declaram as condições de uso de um *software*, tais como permissões, restrições e direitos.
Condições Necessárias:

(i) ***License*** é subclasse de ***OperatingSystemDomainConcept*** já que os Sistemas Operacionais dispõem de no mínimo uma licença.

$$\texttt{License} \sqsubseteq \texttt{OperatingSystemDomainConcept}$$

(ii) Indivíduos da classe ***License*** só relacionam-se através da propriedade ***isAssociatedWith*** com indivíduos da classe ***SoftwareCategory***.

$$\texttt{License} \sqsubseteq \forall\ \texttt{isAssociatedWith}\ .\ \texttt{SoftwareCategory}$$

(iii) Uma licença, tem um criador que pode ser uma pessoa ou uma organização, ou seja, indivíduos da classe **License** só relacionam-se através da propriedade ***hasCreator*** com indivíduos da classe união entre ***Person*** e ***Organization***.

$$\texttt{License} \sqsubseteq \forall\ \texttt{hasCreator}\ (\texttt{Person} \sqcup \texttt{Organization})$$

(iv) Indivíduos da classe ***License*** relacionam-se através da propriedade ***hasCreator*** com pelo menos um indivíduo da classe união entre ***Person*** e ***Organization***.

$$\texttt{Hardware} \sqsubseteq \exists\ \texttt{hasCreator}\ (\texttt{Person} \sqcup \texttt{Organization})$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***License****, pode-se dizer:*

(i) ***isAssociatedWith*** é uma propriedade objeto (Equação A.16) simétrica (Equação A.17), já que se uma licença está associada a uma categoria de *software*, tal categoria também está associada a licença.

$$\texttt{isAssociatedWith} \in \texttt{P}_\texttt{O} \tag{A.16}$$

$$\texttt{isAssociatedWith} \equiv \texttt{isAssociatedWith}^- \tag{A.17}$$

(ii) ***hasCreator*** é uma propriedade objeto, inversa da propriedade *isCreatorOf*.

$$\texttt{hasCreator} \in \texttt{P}_\texttt{O}$$

$$\texttt{hasCreator} \equiv \texttt{isCreatorOf}^-$$

(iii) Cada licença pode ter uma versão. Esta versão pode auxiliar no controle da licença e para isso temos a propriedade ***hasLicenceVersion***. Trata-se de uma subpropriedade de tipo de dados funcional (Equações A.18, A.19 e A.20). Para esta propriedade, o domínio especificado é a classe ***License*** e o contradomínio é **XMLSchema:string** (Equações A.21 e A.22).

$$\texttt{hasLicenseVersion} \in \texttt{P}_\texttt{D} \tag{A.18}$$

$$\texttt{hasLicenseVersion} \sqsubseteq \texttt{version} \tag{A.19}$$

$$\top \sqsubseteq (\leq \texttt{hasLicenseVersion}\ ) \tag{A.20}$$

$$\top \sqsubseteq \forall\, \texttt{hasLicenseVersion}\,.\,\texttt{License}\,(\texttt{License} \neq \bot) \tag{A.21}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasLicenseVersion}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \tag{A.22}$$

Disjunções: A classe ***License*** é disjunta das classes **Organization**, ***Software***, ***Person***, ***Hardware*** e ***SoftwareCategory*** já que estas classes não compartilham instâncias.

$$\texttt{License} \sqsubseteq \begin{pmatrix} \neg\, \texttt{Organization} \sqcap \neg\, \texttt{Software} \sqcap \neg\, \texttt{Person} \sqcap \\ \neg\, \texttt{Hardware} \sqcap \neg\, \texttt{SoftwareCategory} \end{pmatrix}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe ***License***.

$$\{\ \texttt{GNU\_GPL}, \texttt{NokiaOpenSourceLicense} - \texttt{NOKOS}\ \} \sqsubseteq \texttt{License}$$

A seguir é possível observar a relação de uma das instâncias da classe ***License*** com as propriedades P_O e P_D que descrevem a classe.

$$\texttt{isAssociatedWith}\ (\texttt{GNU\_GPL}, \texttt{FreeSoftware})$$

$$\texttt{hasCreator}\ (\texttt{GNU\_GPL}, \texttt{RichardStallman})$$

$$\texttt{hasLicenseVersion}\ (\texttt{GNU\_GPL}, 2.0)$$

- ***Organization:*** esta classe representa as organizações que são formadas por pessoas e podem ser empresas, instituições não empresariais, governamentais, entre outros tipos.
  Condições Necessárias:

  (i) **Organization** é subclasse de **OperatingSystemDomainConcept** já que os Sistemas Operacionais são produzidos e/ou mantidos por organizações.

$$\texttt{Organization} \sqsubseteq \texttt{OperatingSystemDomainConcept}$$

(ii) No contexto do domínio representado as organização tem relação de produção ou desenvolvimento de algum produto, ou seja, assumem a atividade de criação de algum produto. Indivíduos da classe ***Organization*** relacionam-se através da propriedade ***isCreatorOf*** com indivíduos da classe união entre ***Hardware, License, Software*** e ***SoftwareDocumentation***.

$$\texttt{Organization} \sqsubseteq \exists\, \texttt{isCreatorOf} \left(\begin{array}{c} \texttt{Hardware} \sqcup \texttt{License} \sqcup \\ \texttt{Software} \sqcup \texttt{SoftwareDocumentation} \end{array} \right)$$

(iii) Uma organização é formada por pelo menos uma pessoa. Neste contexto, indivíduos da classe **Organization**, relacionam-se através da propriedade **isFormedBy** com pelo menos um indivíduo da classe **Person**.

$$\texttt{Organization} \sqsubseteq \exists\, \texttt{isFormedBy} \,.\, \texttt{Person}$$

Propriedades - *Sobre as propriedades que descrevem a classe **Organization**, pode-se dizer:*

(i) Ambas as propriedades **isCreatorOf** e **isFormedBy** são propriedades objeto (Equações A.23 e A.24). **isCreatorOf** é uma subpropriedade inversa de **hasCreator** (Equações A.25 e A.26).

$$\texttt{isCreatorOf} \in \texttt{P}_0 \tag{A.23}$$
$$\texttt{isFormedBy} \in \texttt{P}_0 \tag{A.24}$$
$$\texttt{isCreatorOf} \sqsubseteq \texttt{hasActivity} \tag{A.25}$$
$$\texttt{isCreatorOf} \equiv \texttt{hasCreator}^- \tag{A.26}$$

Disjunções: - *Sobre as classes disjuntas de **Organization**, pode-se dizer:*

(i) A classe **Organization** é disjunta das classes **Software**, **License**, **Person**, **Hardware** e **SoftwareCategory** já que estas classes não compartilham instâncias.

$$\texttt{Organization} \sqsubseteq \left(\begin{array}{c} \neg\, \texttt{Software} \sqcap \neg\, \texttt{License} \sqcap \neg\, \texttt{SoftwareCategory} \sqcap \\ \neg\, \texttt{Hardware} \sqcap \neg\, \texttt{Person} \end{array} \right)$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **Organization**.

$$\{\, \texttt{GNU\_Project}, \texttt{Microsoft} \,\} \sqsubseteq \texttt{Organization}$$

A seguir é possível observar a relação de uma das instâncias da classe **Organization** com as propriedades P_O que a descrevem.

$$\texttt{isCreatorOf}\ (\texttt{Microsoft}, \texttt{MS} - \texttt{DOS})$$
$$\texttt{isFormedBy}\ (\texttt{Microsoft}, \texttt{BillGates})$$

- ***VirtualCommunity:*** esta classe representa um tipo de organização disponível à seus participantes de forma virtual, ou seja, na *internet*. Este tipo de organização é popularmente conhecida como comunidades virtuais e podem agrupar pessoas com os mais variados interesses, como por exemplo, pessoas que compartilham informações sobre sistemas operacionais.
  Condições Necessárias:

  (i) **VirtualCommunity** é subclasse de **Organization**.

$$\texttt{VitualCommunity} \sqsubseteq \texttt{Organization}$$

(ii) As comunidades virtuais tem interesse nos mais variados assuntos. Indivíduos desta classe só relacionam-se através da propriedade ***hasInterestIn*** com indivíduos da classe ***owl:Thing***

$$\texttt{VitualCommunity} \sqsubseteq \forall\, \texttt{hasInterestIn} \,.\, \texttt{owl:Thing}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***VirtualCommunity****, pode-se dizer:*

(i) **hasInterestIn** é uma propriedade objeto que tem a classe **VirtualCommunit** como domínio.

$$\texttt{hasInterestIn} \in \texttt{P}_\texttt{O} \tag{A.27}$$

$$\top \sqsubseteq \forall\, \texttt{hasInterestIn} \,.\, \texttt{VirtualCommunity} \tag{A.28}$$

- ***Person:*** classe que representa o conjunto dos seres humanos.

Condição Necessária:

(i) **Person** é subclasse de **OperatingSystemDomainConcept** já que as pessoas são agentes que podem interagir com os Sistemas Operacionais. Essa interação depende do tipo de papel assumido pela pessoa. Por exemplo, uma pessoa pode ser um usuário do sistema.

$$\texttt{Person} \sqsubseteq \texttt{OperatingSystemDomainConcept}$$

Propriedade - *Sobre a propriedade que descreve a classe* ***Person****, pode-se dizer:*

(i) Toda pessoa tem um nome. A propriedade **personName** é de tipo de dados e funcional (Equações A.29 e A.30). Para esta propriedade, o domínio especificado é a classe **Person** (Equação A.31) e o contradomínio é **XMLSchema:string** (Equação A.32).

$$\texttt{personName} \in \texttt{P}_\texttt{D} \tag{A.29}$$

$$\top \sqsubseteq (\leq\ 1\ \texttt{personName}\ ) \tag{A.30}$$

$$\top \sqsubseteq \forall\, \texttt{personName} \,.\, \texttt{Person}\ (\texttt{Person} \neq \bot) \tag{A.31}$$

$$\top \sqsubseteq \forall\, \texttt{personName}^{-} \,.\, \texttt{XMLSchema:string}\ (\texttt{XMLSchema:string} \neq \bot) \tag{A.32}$$

Disjunções: - *Sobre as classes disjuntas de* ***Person****, pode-se dizer:*

(i) A classe **Person** é disjunta das classes **Software**, **License**, **Organization**, **Hardware** e **SoftwareCategory** já que estas classes não compartilham instâncias.

$$\texttt{Person} \sqsubseteq \begin{pmatrix} \neg\, \texttt{Software} \sqcap \neg\, \texttt{License} \sqcap \neg\, \texttt{Organization} \sqcap \\ \neg\, \texttt{Hardware} \sqcap \neg\, \texttt{SoftwareCategory} \end{pmatrix}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **Person**.

$$\{\, \texttt{RichardStallman}, \texttt{BillGates}\, \} \sqsubseteq \texttt{Person}$$

As próximas seções detalham as subclasses de **Person** que são ***User***, ***Developer***, ***Approver***, ***Author***, ***Maintainer*** e ***Reviser***. Estas subclasses não são disjuntas já que instâncias podem ser compartilhadas entre as classes. Por exemplo, a instância ***Virginia*** participa da

classe ***Reviser*** e ***Approver*** conforme demonstrado pela figura A.2 extraída do Protégé. A Figura A.3 introduz as subclasses. As classes representadas por um circulo contém condições necessárias e suficientes. As classes representadas por um quadrado contém apenas condições necessárias.

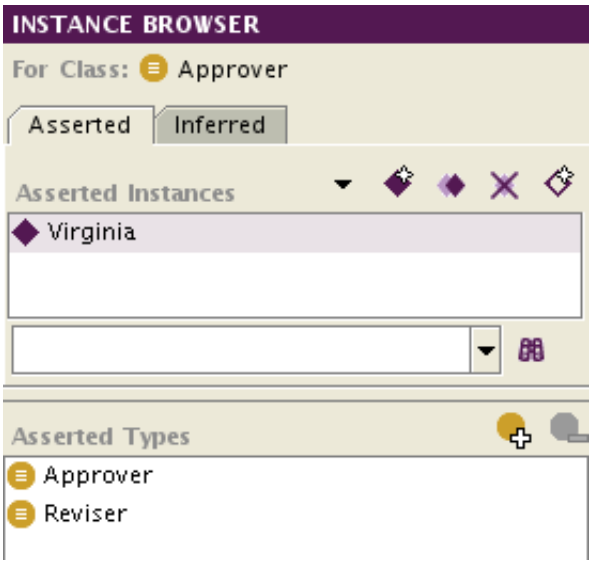

Figura A.2: Uma instância da classe *Approver* e *Reviser*

- ***User:*** esta classe representa os usuários de um *software*, da documentação de um *software* ou de um *hardware* e que também podem participar de uma comunidade virtual.
  Condições Necessárias:

  (i) **User** é subclasse de **Person** já que todo usuário é uma pessoa.

  $$\texttt{User} \sqsubseteq \texttt{Person}$$

  (ii) Indivíduos da classe ***User*** só relacionam-se através da propriedade ***contributesWithDocumentation*** com indivíduos da classe ***SoftwareDocumentation***.

  $$\texttt{User} \sqsubseteq \forall\, \texttt{contributesWithDocumentation}\,.\,\texttt{SoftwareDocumentation}$$

  (iii) Indivíduos da classe ***User*** só relacionam-se através da propriedade ***participatesIn*** com indivíduos da classe ***VitualCommunity***.

  $$\texttt{User} \sqsubseteq \forall\, \texttt{participatesIn}\,.\,\texttt{VirtualCommunity}$$

  (iv) Indivíduos da classe **User** relacionam-se através da propriedade ***uses*** com pelo menos um indivíduo da classe resultante da união entre ***Hardware, Software*** e ***SoftwareDocumentation***.

  $$\texttt{User} \sqsubseteq \exists\, \texttt{uses} \left(\begin{array}{c} \texttt{Hardware} \sqcup \texttt{Software} \\ \sqcup\, \texttt{SoftwareDocumentation}) \end{array} \right)$$

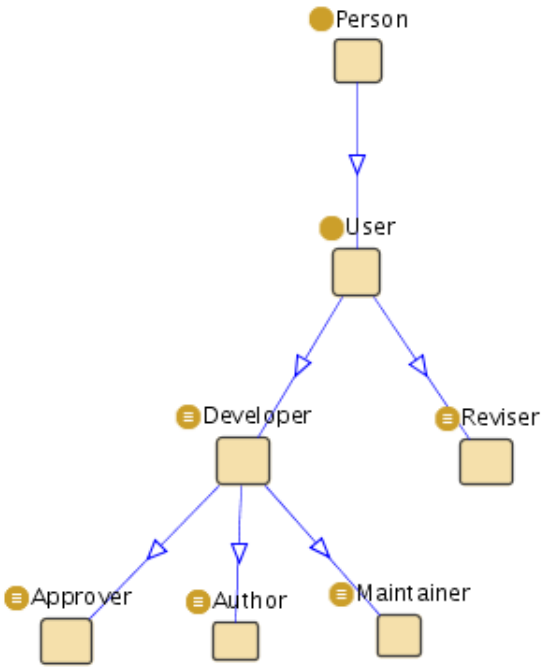

Figura A.3: A classe *Person* e suas subclasses

Propriedades - *Sobre as propriedades que descrevem a classe* ***User****, pode-se dizer:*

(i) **contributesWithDocumentation** é uma propriedade objeto (Equação A.33) cujo domínio é **User** e o contradomínio é **SoftwareDocumentation** (Equações A.34 e A.35).

$$\texttt{contributesWithDocumentation} \in \texttt{P}_0 \tag{A.33}$$

$$\top \sqsubseteq \forall\, \texttt{contributesWithDocumentation} \,.\, \texttt{User} \tag{A.34}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{contributesWithDocumentation}^- \,. \\ \texttt{SoftwareDocumentation} \end{pmatrix} \tag{A.35}$$

(ii) **participatesIn** é uma propriedade objeto (Equação A.36) cujo domínio é **User** e o contradomínio é **VirtualCommunity** (Equação A.37 e A.38).

$$\texttt{participatesIn} \in \texttt{P}_0 \tag{A.36}$$

$$\top \sqsubseteq \forall\, \texttt{participatesIn} \,.\, \texttt{User} \tag{A.37}$$

$$\top \sqsubseteq \forall\, \texttt{participatesIn}^- \,.\, \texttt{VitualCommunity} \tag{A.38}$$

(iii) **uses** é uma subpropriedade objeto.

$$\texttt{uses} \in \texttt{P}_0$$

$$\texttt{uses} \sqsubseteq \texttt{hasActivity}$$

A figura A.4 mostra a propriedade *hasActivity* e suas subpropriedades.

(iv) Os usuários podem ser identificados por um nome. É bem comum os usuários associarem apelidos no lugar do nome real. A propriedade **hasUserName** representa a relação de um usuário e seu respectivo nome. Trata-se de uma propriedade de tipo

Figura A.4: A propriedade *hasActivity* e suas subpropriedades

de dados funcional (Equações A.39 e A.40). Para esta propriedade, o domínio especificado é a classe **User** e o contradomínio é **XMLSchema:string** (Equações A.41 e A.42).

$$\texttt{hasUserName} \in \texttt{P}_\texttt{D} \quad \text{(A.39)}$$

$$\top \sqsubseteq (\leq 1\ \texttt{hasUserName}) \quad \text{(A.40)}$$

$$\top \sqsubseteq \forall\, \texttt{hasUserName} \,.\, \texttt{User} \quad (\texttt{User} \neq \bot) \quad \text{(A.41)}$$

$$\top \sqsubseteq \forall\, \texttt{hasUserName}^{-} \,.\, \texttt{XMLSchema:string} \quad (\texttt{XMLSchema:string} \neq \bot) \quad \text{(A.42)}$$

(v) Alguns *softwares* controlam o acesso de seus usuários, como por exemplo, os sistemas operacionais. Para tanto, os usuários devem dispor de uma senha. ***hasUserPassword*** representa a relação de um usuário e sua respectiva senha e é uma propriedade de tipo de dados funcional (Equações A.43 e A.44). Para esta propriedade, o domínio é a classe **User** e o contradomínio é **XMLSchema:string** (Equações A.45 e A.46).

$$\texttt{hasUserPassword} \in \texttt{P}_\texttt{D} \quad \text{(A.43)}$$

$$\top \sqsubseteq (\leq 1\ \texttt{hasUserPassword}) \quad \text{(A.44)}$$

$$\top \sqsubseteq \forall\, \texttt{hasUserPassword} \,.\, \texttt{User} \quad (\texttt{User} \neq \bot) \quad \text{(A.45)}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasUserPassword}^{-} \,.\, \texttt{XMLSchema:string} \\ (\texttt{XMLSchema:string} \neq \bot) \end{pmatrix} \quad \text{(A.46)}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **User**.

$$\{\ \texttt{Afonso}, \texttt{RodrigoBraga}\ \} \sqsubseteq \texttt{User}$$

A seguir é possível observar a relação de uma das instâncias da classe **User** com as propriedades P_O e P_D que a descrevem.

$$\texttt{contributesWithDocumentation}\ (\texttt{RodrigoBraga}, \texttt{SwTOOWLDOC})$$
$$\texttt{contributesWithDocumentation}\ (\texttt{RodrigoBraga}, \texttt{OSOntoOWLDOC})$$
$$\texttt{contributesWithDocumentation}\ (\texttt{RodrigoBraga}, \texttt{SwTOiOWLDOC})$$
$$\texttt{uses}\ (\texttt{RodrigoBraga}, \texttt{SwTO})$$
$$\texttt{uses}\ (\texttt{RodrigoBraga}, \texttt{SwTOi})$$
$$\texttt{uses}\ (\texttt{RodrigoBraga}, \texttt{OSOnto})$$
$$\texttt{hasUserName}(\texttt{RodrigoBraga}, \texttt{rbraga})$$
$$\texttt{userPassword}(\texttt{RodrigoBraga}, \texttt{123abc})$$

- ***Developer:*** esta classe representa os usuários desenvolvedores que implementam programas de computador para que atendam os requisitos estabelecidos.
Condição Necessária e Suficiente:

  (i) Indivíduos da classe ***Developer*** só relacionam-se através da propriedade ***implements*** com indivíduos da classe ***Software*** se e somente se indivíduos que implementam *software* forem da classe ***Developer***.

$$\texttt{Developer} \equiv \forall\ \texttt{implements}\ .\ \texttt{Software}$$

Condições Necessárias:

  (i) **Developer** é subclasse de **User** já que todo desenvolvedor é um usuário do *software* em desenvolvimento.

$$\texttt{Developer} \sqsubseteq \texttt{User}$$

  (ii) Os desenvolvedores podem criar a documentação do *software* que eles implementam. Desta forma, indivíduos da classe ***Developer*** só relacionam-se através da propriedade ***isCreatorOf*** com indivíduos da classe ***SoftwareDocumentation***.

$$\texttt{Developer} \sqsubseteq \forall\ \texttt{isCreatorOf}\ .\ \texttt{SoftwareDocumentation}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***Developer****, pode-se dizer:*

  (i) **implements** é uma subpropriedade objeto.

$$\texttt{implements} \in \texttt{P}_0$$
$$\texttt{implements} \sqsubseteq \texttt{hasActivity}$$

  (ii) A outra propriedade que descreve a classe **Developer** é **isCreatorOf**, discutida anteriormente na classe **Organization**.

Instanciação: O exemplo a seguir demonstra instâncias da classe **Developer**.

$$\{\ \texttt{DaveJones}, \texttt{TimGardner}\ \} \sqsubseteq \texttt{Developer}$$

Ambas as instâncias mensionadas acima representam pessoas que fazem parte da árvore ativa de desenvolvimento do Linux. A seguir é possível observar a relação de uma das instâncias da classe **Developer** com as propriedades P_O e P_D que a descrevem.

$$\texttt{implements}(\texttt{DaveJones}, \texttt{Linux})$$
$$\texttt{isCreatorOf}(\texttt{DaveJones}, \texttt{xxx})$$

- ***Approver:*** esta classe representa os usuários com habilidades técnicas e permissões para aprovar um *software* ou uma documentação sobre o *software*.
Condição Necessária e Suficiente:

  (i) Indivíduos da classe ***Approver*** só relacionam-se através da propriedade ***approves*** com indivíduos da classe união resultante entre ***Software*** e ***SoftwareDocumentation*** se e somente se todo softwar ou documentação for aprovado por um indivíduo da classe ***Approver***.

$$\texttt{Approver} \equiv \forall\ \texttt{approves}\ (\ \texttt{Software} \sqcup \texttt{SoftwareDocumentation})$$

Condições Necessárias:

(i) **Approver** é subclasse de **Developer** já que o aprovador também é um desenvolvedor e pode assumir a liderança de um projeto.

$$\texttt{Approver} \sqsubseteq \texttt{Developer}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***Approver****, pode-se dizer:*

(i) **approves** é uma subpropriedade objeto.

$$\texttt{approves} \in \texttt{P}_0$$
$$\texttt{approves} \sqsubseteq \texttt{hasActivity}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **Approver** bem como a relação da instância com a propriedade P_O que a descrevem.

$$\{\ \texttt{AndrewMorton}\ \} \sqsubseteq \texttt{Approver}$$
$$\texttt{approves}(\texttt{AndrewMorton}, [\texttt{PATCH}]\texttt{x86}) \{\ \texttt{Virginia}\ \} \sqsubseteq \texttt{Approver}$$
$$\texttt{approves}(\texttt{Virginia}, \texttt{OSOnto})$$
$$\texttt{approves}(\texttt{Virginia}, \texttt{SwTO})$$
$$\texttt{approves}(\texttt{Virginia}, \texttt{SwTOi})$$

- ***Author:*** esta classe representa os desenvolvedores criadores de um *software*.

Condição Necessária e Suficiente:

(i) Indivíduos da classe ***Author*** só relacionam-se através da propriedade ***isCreatorOf*** com indivíduos da classe ***Software*** se e somente se todo software for criado por um indivíduo da classe ***Author***.

$$\texttt{Author} \equiv \forall\ \texttt{isCreatorOf}\ .\ \texttt{Software}$$

Condições Necessárias:

(i) **Author** é subclasse de **Developer** já que todo author de um *software* também é um desenvolvedor.

$$\texttt{Author} \sqsubseteq \texttt{Developer}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***Author****, pode-se dizer:*

(i) A propriedade que descreve a classe **Author** é **isCreatorOf**, discutida anteriormente na classe **Organization**.

Instanciação: O exemplo a seguir demonstra instâncias da classe **Author**.

$$\{\ \texttt{LinusTorvalds}, \texttt{GuidoVanRossum}, \texttt{Daniella}, \texttt{JoroenWijnhout}\ \} \sqsubseteq \texttt{Author}$$

A seguir é possível observar a relação das instâncias da classe **Author** com a propriedade P_O que a define.

$$\texttt{isCreatorOf}(\texttt{LinusTorvalds}, \texttt{Linux})$$
$$\texttt{isCreatorOf}(\texttt{Daniella}, \texttt{SwTOi})$$
$$\texttt{isCreatorOf}(\texttt{JoroenWijnhout}, \texttt{Kile})$$
$$\texttt{isCreatorOf}(\texttt{GuidoVanRossum}, \texttt{PythonScriptingLanguage})$$

- ***Maintainer:*** são desenvolvedores responsáveis em coordenar a atividade ou contribuição de outros desenvolvedores na implementação do *software*.
Condição Necessária e Suficiente:

  (i) Indivíduos da classe ***Maintainer*** só relacionam-se através da propriedade ***coordinates*** com indivíduos da classe ***Developer*** se e somente se os desenvolvedores forem coordenados por indivíduos da classe ***Maintainer***.

$$\texttt{Maintainer} \equiv \forall\, \texttt{coordinates}\,.\,\texttt{Developer}$$

  Condições Necessárias:

  (i) **Maintainer** é subclasse de **Developer** já que todo mantenedor também é um desenvolvedor.

$$\texttt{Maintainer} \sqsubseteq \texttt{Developer}$$

  Propriedades - *Sobre as propriedades que descrevem a classe* ***Maintainer****, pode-se dizer:*

  (i) **coordinates** é uma subpropriedade objeto.

$$\texttt{coordinates} \in \texttt{P}_\texttt{O}$$
$$\texttt{coordinates} \sqsubseteq \texttt{hasActivity}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **Maintainer**.

$$\{\, \texttt{LinusTorvalds}, \texttt{DavidMiller} \,\} \sqsubseteq \texttt{Maintainer}$$

  A seguir é possível observar a relação das duas instâncias da classe **Maintainer** com a propriedade P_O que a descreve.

$$\texttt{coordinates}(\texttt{LinusTorvalds}, \texttt{DavidMiller})$$
$$\texttt{coordinates}(\texttt{DavidMiller}, \texttt{DaveJones})$$
$$\texttt{coordinates}(\texttt{DavidMiller}, \texttt{TimGardner})$$

- ***Reviser:*** esta classe representa o grupo de usuários responsáveis pela revisão do *software* ou de sua documentação.
Condição Necessária e Suficiente:

  (i) Indivíduos da classe ***Reviser*** relacionam-se através da propriedade ***reviews*** com pelo menos um indivíduos da classe resultante entre a união de ***Software*** e ***SoftwareDocumentation*** se e somente se individuos desta classe união forem revisados por pelo menos um indivíduo da classe ***Reviser***.

$$\texttt{Reviser} \equiv \exists\, \texttt{reviews}\,(\texttt{Software} \sqcup \texttt{SoftwareDocumentation})$$

  Condições Necessárias:

  (i) **Reviser** é subclasse de **User** já que todo revisor é um usuário do software ou da documentação do software.

$$\texttt{Reviser} \sqsubseteq \texttt{User}$$

  Propriedades - *Sobre as propriedades que descrevem a classe* ***Reviser****, pode-se dizer:*

(i) **reviews** é uma subpropriedade objeto.

$$\texttt{reviews} \in \texttt{P}_\texttt{O}$$
$$\texttt{reviews} \sqsubseteq \texttt{hasActivity}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **Reviser**.

$$\{\ \texttt{Afonso}\ \} \sqsubseteq \texttt{Reviser}$$

A seguir é possível observar a relação de uma instância da classe **Maintainer** com a propriedade P_O que a descreve.

$$\texttt{reviews}(\texttt{Afonso}, \texttt{OSOnto})$$
$$\texttt{reviews}(\texttt{Afonso}, \texttt{OSOntoOWLDOC})$$

- ***Software:*** esta classe representa o conjunto de programas que são implementados em uma linguagem computacional para qual existe uma máquina computacional.
Condições Necessárias:

(i) **Software** é subclasse de **OperatingSystemDomainConcept** por ser um conceito associado ao domínio de sistema operacional.

$$\texttt{Software} \sqsubseteq \texttt{OperatingSystemDomainConcept}$$

(ii) Indivíduos da classe ***Software*** só relacionam-se através da propriedade ***hasCategory*** com indivíduos da classe ***SoftwareCategory***.

$$\texttt{Software} \sqsubseteq \forall\ \texttt{hasCategory}\ .\ \texttt{SoftwareCategory}$$

(iii) Indivíduos da classe ***Software*** relacionam-se através da propriedade ***hasCategory*** com pelo menos um indivíduo da classe **SoftwareCategory**.

$$\texttt{Software} \sqsubseteq \exists\ \texttt{hasCategory}\ .\ \texttt{SoftwareCategory}$$

(iv) Indivíduos da classe ***Software*** relacionam-se através da propriedade ***hasCreator*** com pelo menos um indivíduo da classe resultante entre a união de ***Person*** e ***Organization***.

$$\texttt{Software} \sqsubseteq \exists\ \texttt{hasCreator}\ (\texttt{Person} \sqcup \texttt{Organization})$$

(v) Indivíduos da classe ***Software*** só relacionam-se através da propriedade ***hasLicense*** com indivíduos da classe ***License***.

$$\texttt{Software} \sqsubseteq \forall\ \texttt{hasLicense}\ .\ \texttt{License}$$

(vi) Indivíduos da classe ***Software*** relacionam-se através da propriedade ***hasLicense*** com no mínimo um indivíduo da classe ***License***.

$$\texttt{Software} \sqsubseteq \geq\ \texttt{hasLicense}\ .\ 1$$

(vii) Um *software* pode ser composto por outro *software*. Desta forma, indivíduos da classe ***Software*** só relacionam-se através da propriedade ***isComposedOf*** com indivíduos

da classe ***Software***.

$$\texttt{Software} \sqsubseteq \forall\, \texttt{isComposedOf} \,.\, \texttt{Software}$$

(viii) Indivíduos da classe ***Software*** só relacionam-se através da propriedade ***isDocumentedBy*** com indivíduos da classe ***SoftwareDocumentation***.

$$\texttt{Software} \sqsubseteq \forall\, \texttt{isDocumentedBy} \,.\, \texttt{SoftwareDocumentation}$$

(ix) Indivíduos da classe ***Software*** relacionam-se através da propriedade ***isDocumentedBy*** com pelo menos um indivíduo da classe ***SoftwareDocumentation***.

$$\texttt{Software} \sqsubseteq \exists\, \texttt{isDocumentedBy} \,.\, \texttt{SoftwareDocumentation}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***Software****, pode-se dizer:*

(i) **hasCategory** é uma propriedade objeto funcional (Equações A.47 e A.48), cujo domínio é a classe **Software** e o contradomínio é **SoftwareCategory** (Equações A.49 e A.50).

$$\texttt{hasCategory} \in \texttt{P}_0 \tag{A.47}$$
$$\top \sqsubseteq (\leq\, 1\, \texttt{hasCategory}) \tag{A.48}$$
$$\top \sqsubseteq \forall\, \texttt{hasCategory} \,.\, \texttt{Software} \tag{A.49}$$
$$\top \sqsubseteq \forall\, \texttt{hasCategory}^{-} \,.\, \texttt{SoftwareCategory} \tag{A.50}$$

(ii) **hasLicense** é uma propriedade objeto funcional (Equações A.51 e A.52) cujo domínio é a classe **Software** e o contradomínio é a classe **License** (Equações A.53 e A.54).

$$\texttt{hasLicense} \in \texttt{P}_0 \tag{A.51}$$
$$\top \sqsubseteq (\leq\, 1\, \texttt{hasLicense}) \tag{A.52}$$
$$\top \sqsubseteq \forall\, \texttt{hasLicense} \,.\, \texttt{Software} \tag{A.53}$$
$$\top \sqsubseteq \forall\, \texttt{hasLicense}^{-} \,.\, \texttt{License} \tag{A.54}$$

(iii) **isDocumentedBy** é uma propriedade objeto, inversa a ***documents*** (Equações A.55 e A.56) cujo domínio é a classe **Software** e o contradomínio é a classe **SoftwareDocumentation** (Equações A.57 e A.58).

$$\texttt{isDocumentedBy} \in \texttt{P}_0 \tag{A.55}$$
$$\texttt{isDocumentedBy} \equiv \texttt{documents}^{-} \tag{A.56}$$
$$\top \sqsubseteq \forall\, \texttt{isDocumentedBy} \,.\, \texttt{Software} \tag{A.57}$$
$$\top \sqsubseteq \forall\, \texttt{isDocumentedBy}^{-} \,.\, \texttt{SoftwareDocumentation} \tag{A.58}$$

(iv) Também estão associadas a classe **Software** as propriedades objeto **hasCreator** (explicada anteriormente na classe **License**) e **isComposedOf** (explicada na classe **Hardware**).

(v) Cada *software* dispõe de uma versão que pode auxiliar no controle do *software*. **hasSoftwareVersion** é uma subpropriedade de tipo de dados (Equações A.59 e A.60) funcional (Equação A.61). Para esta propriedade, o domínio especificado é a classe

Software (Equação A.62) e o contradomínio é **XMLSchema:string** (Equação A.63).

$$\texttt{hasSoftwareVersion} \in \texttt{P}_\texttt{D} \tag{A.59}$$
$$\texttt{hasSoftwareVersion} \sqsubseteq \texttt{version} \tag{A.60}$$
$$\top \sqsubseteq (\leq \texttt{1icenseVersion}\ ) \tag{A.61}$$
$$\top \sqsubseteq \forall\, \texttt{licenseVersion}\,.\,\texttt{License}\ (\texttt{License} \neq \bot) \tag{A.62}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{licenseVersion}^{-}\,.\,\texttt{XMLSchema:string} \\ (\texttt{XMLSchema:string} \neq \bot) \end{pmatrix} \tag{A.63}$$

Disjunções: A classe **Software** é disjunta das classes **Person**, **Hardware**, **License**, **Organization** e **SoftwareCategory** já que estas classes não compartilham instâncias.

$$\texttt{Software} \sqsubseteq \begin{pmatrix} \neg\,\texttt{Hardware} \sqcap \neg\,\texttt{License} \sqcap \\ \neg\,\texttt{Person} \sqcap \neg\,\texttt{Organization} \sqcap \neg\,\texttt{SoftwareCategory} \end{pmatrix}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **Software**.

$$\{\ \texttt{Protege}, \texttt{Kile}, \texttt{Amsn}, \texttt{Skype}\ \} \sqsubseteq \texttt{Software}$$

A seguir é possível observar a relação de uma das instância da classe **Software** com as propriedades P_O e P_D que a descrevem.

$$\texttt{hasCategory}(\texttt{Kile}, \texttt{FreeSoftware})$$
$$\texttt{hasCreator}(\texttt{Kile}, \texttt{JoroenWijnhout})$$
$$\texttt{hasLicense}(\texttt{Kile}, \texttt{GNUGPL})$$
$$\texttt{isComposedOf}(\texttt{Kile}, \texttt{LaTeX})$$
$$\texttt{isDocumentedBy}(\texttt{Kile}, \texttt{KileHandbook})$$
$$\texttt{hasSoftwareVersion}(\texttt{Kile}, 1.8.1)$$

A Figura A.5 introduz as subclasses de *Software*.

- ***DevelopmentSoftware:*** esta classe representa os softwares que são usados no processo de desenvolvimento de outros softwares.
  Condições Necessárias:

  (i) **DevelopmentSoftware** é subclasse de **Software**.

  $$\texttt{DevelopmentSoftware} \sqsubseteq \texttt{Software}$$

  Disjunções: *Sobre as classes disjuntas de* ***DevelopmentSoftware****, pode-se dizer:*

  (i) A classe **DevelopmentSoftware** é disjunta das classes **NetworkSoftwere, Ontology, SystemProgram, SystemSoftware** e *UserInterface* já que estas classes não compartilham instâncias.

  $$\texttt{DevelopmentSoftware} \sqsubseteq \begin{pmatrix} \neg\,\texttt{NetworkSoftware} \sqcap \neg\,\texttt{Ontology} \sqcap \neg\,\texttt{SystemProgram} \sqcap \\ \neg\,\texttt{SystemSoftware} \sqcap \neg\,\texttt{UserInterface} \end{pmatrix}$$

  A Figura A.6 introduz as subclasses de *DevelopmentSoftware*. Todas elas são disjuntas entre si.

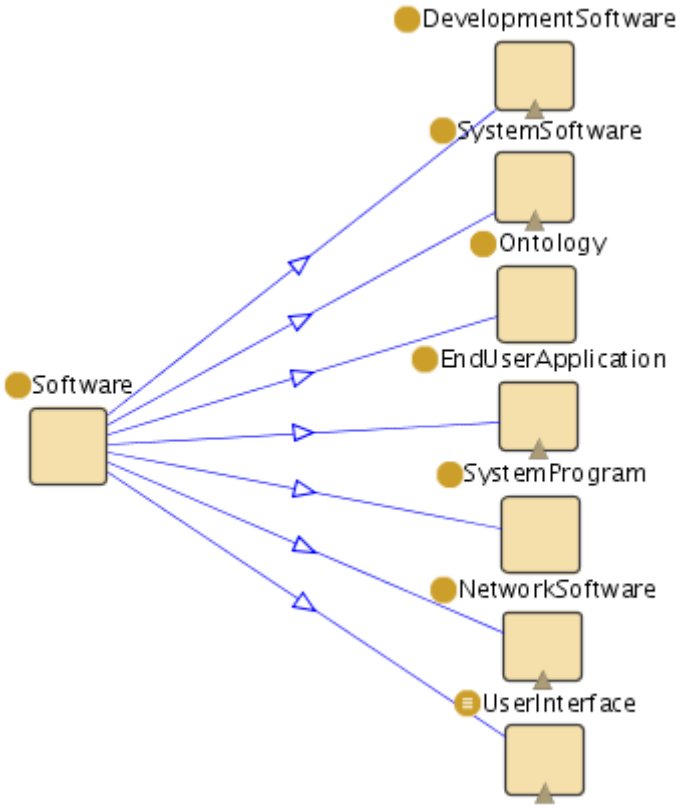

Figura A.5: A classe *Software* e suas subclasses

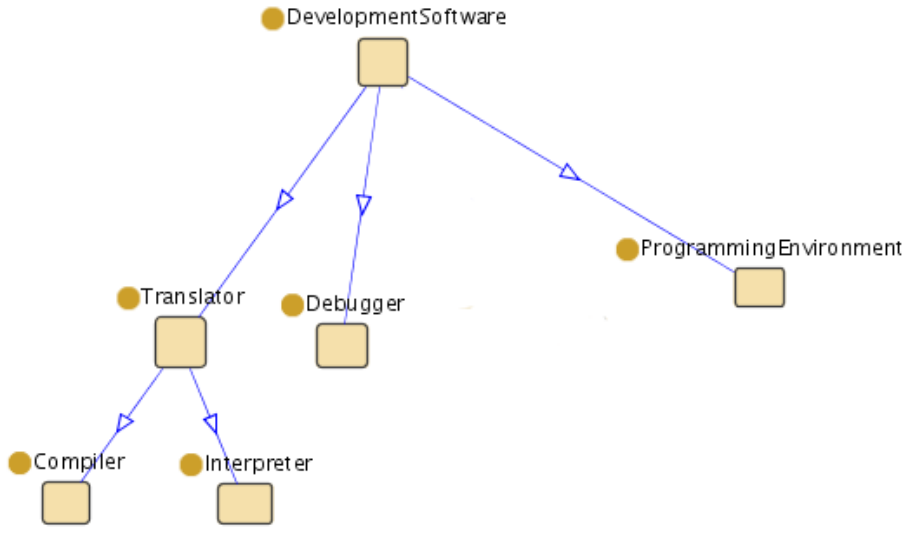

Figura A.6: A classe *DevelopmentSoftware* e suas subclasses

- ***Debugger:*** esta classe representa os depuradores que são programas usados para visualizar os caminhos percorridos por um programa durante sua

execução. Este tipo de programa possibilita diagnosticar possíveis erros indicados por ações esperadas do sistema que não ocorrem.
Condições Necessárias:

(i) **Debugger** é subclasse de **DevelopmentSoftware**.

$$\texttt{Debugger} \sqsubseteq \texttt{DevelopmentSoftware}$$

Instanciação: O exemplo a seguir demonstra uma possível instância da classe **Debugger**.

$$\{\ \texttt{Gdb}\ \} \sqsubseteq \texttt{Debbuger}$$

- ***ProgrammingEnvironment:*** trata-se de um conjunto de programas que viabilizam o desenvolvimento de outros *softwares* por meio de uma linguagem de programação.
Condições Necessárias:

(i) **ProgrammingEnvironment** é subclasse de **DevelopmentSoftware**.

$$\texttt{ProgrammingEnvironment} \sqsubseteq \texttt{DevelopmentSoftware}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **ProgrammingEnvironment**.

$$\{\ \texttt{Eclipse}, \texttt{JBuilder}\ \} \sqsubseteq \texttt{ProgrammingEnvironment}$$

- ***Translator:*** esta classe representa o conjunto dos programas que traduzem o código-fonte de um software em um formato de dados que pode ser executado pelo computador.
Condições Necessárias:

(i) **Translator** é subclasse de **DevelopmentSoftware**.

$$\texttt{Translator} \sqsubseteq \texttt{DevelopmentSoftware}$$

As classes disjuntas *Compiler* e *Interpreter* são subclasses de *Translator*.

- ***Compiler:*** esta classe representa os compiladores que são responsáveis em reunir o código-fonte de um software, bem como os componentes necessários à sua execução escrito em uma linguagem computacinal, de forma que o computador possa executar o software compilado.
Condições Necessárias:

(i) **Compiler** é subclasse de **EndUserApplication**.

$$\texttt{Compiler} \sqsubseteq \texttt{EndUserApplication}$$

(ii) Todo compilador suporta uma linguagem de programação.

$$\texttt{Compiler} \sqsubseteq \forall\ \texttt{supports}\ .\ \texttt{ProgrammingLanguage}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***Compiler****, pode-se dizer:*

(i) **supports** é uma propriedade objeto, cujo domínio é a classe **Compiler** e o contra-domínio é a classe **ProgrammingLanguage**.

$$\texttt{supports} \in \texttt{P}_0$$
$$\top \sqsubseteq \forall\, \texttt{supports} \,.\, \texttt{Compiler}$$
$$\top \sqsubseteq \forall\, \texttt{supports}^{-} \,.\, \texttt{ProgrammingLanguage}$$

Instanciação: O exemplo a seguir demonstra uma possível instância da classe **Compiler**.

$$\{\, \texttt{Gcc} \,\} \sqsubseteq \texttt{Compiler}$$

- ***Interpreter:*** esta classe representa os programas de computador que lêem um código-fonte de um software e os convertem em código executável. Seu funcionamento pode variar de acordo com a implementação. Em alguns casos o interpretador lê linha-a-linha e converte em código objeto à medida que vai executando o programa.
  Condições Necessárias:

  (i) **Interpreter** é subclasse de **Translator**.

  $$\texttt{Interpreter} \sqsubseteq \texttt{Translator}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **Interpreter**.

  $$\{\, \texttt{PerlInterpreter}, \texttt{PHPInterpreter}, \texttt{PythonInterpreter} \,\} \sqsubseteq \texttt{Interpreter}$$

- ***EndUserApplication:*** esta classe representa o conjunto das aplicações que permitem a seus usuários realizarem uma ou mais tarefas específicas, como por exemplo, editar um texto. As aplicações representam uma camada intermediária entre os usuários e um outro nível de *software* conhecido como **SystemSoftware** que por sua vez, está mais próximo do *hardware*. Existe uma forte relação entre as aplicações e o *system software*. Se o *system software* não existisse, uma simples atividade como, por exemplo, escrever um programa "HelloWorld.java"iria exigir do programador que ele implementasse requisições ao *hardware* em baixo nível. Freqüentemente as aplicações também são chamadas de *user-space programs*.
  Condições Necessárias:

  (i) **EndUserApplication** é subclasse de **Software**.

  $$\texttt{EndUserApplication} \sqsubseteq \texttt{Software}$$

  (ii) Indivíduos da classe *EndUserApplication* só relacionam-se através da propriedade *interactsWith* com indivíduos da classe *SystemSoftware*.

  $$\texttt{EndUserApplication} \sqsubseteq \forall\, \texttt{interactsWith} \,.\, \texttt{SystemSoftware}$$

  (iii) Indivíduos da classe *EndUserApplication* relacionam-se através da propriedade *interactsWith* com pelo menos um indivíduo da classe *SystemSoftware*.

  $$\texttt{EndUserApplication} \sqsubseteq \exists\, \texttt{interactsWith} \,.\, \texttt{SystemSoftware}$$

  (iv) Indivíduos da classe *EndUserApplication* só relacionam-se através da propriedade *ha-*

sUserInterface com indivíduos da classe *UserInterface*.

$$\texttt{EndUserApplication} \sqsubseteq \forall\, \texttt{hasUserInterface} \,.\, \texttt{UserInterface}$$

(v) Indivíduos da classe *EndUserApplication* relacionam-se através da propriedade *hasUserInterface* com pelo menos um indivíduos da classe *UserInterface*.

$$\texttt{EndUserApplication} \sqsubseteq \exists\, \texttt{hasUserInterface} \,.\, \texttt{UserInterface}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***EndUserApplication****, pode-se dizer:*

(i) **hasUserInterface** é uma propriedade objeto funcional (Equações A.64 e A.65).

$$\texttt{hasUserInterface} \in \texttt{P}_0 \tag{A.64}$$

$$\top \sqsubseteq (\leq\ 1\ \texttt{hasUserInterface}) \tag{A.65}$$

(ii) Também está associada a classe **EndUserApplication** a propriedades objeto **interactsWith** (explicada anteriormente na classe **Hardware**).

Disjunções: A classe **EndUserApplication** é disjunta das classes **Ontology** e **SystemSoftware** já que estas classes não compartilham instâncias.

$$\texttt{EndUserApplication} \sqsubseteq \neg\, \texttt{SystemSoftware} \sqcap \neg\, \texttt{Ontology}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **EndUserApplication**.

$$\{\ \texttt{xpdf}, \texttt{Gimp}, \texttt{XMMS}\ \} \sqsubseteq \texttt{EndUserApplication}$$

A seguir é possível observar a relação de uma das instância da classe **EndUserApplication** com as propriedades P_O e P_D que a descrevem.

$$\texttt{hasUserInterface}(\texttt{XMMS}, \texttt{Gnome})$$
$$\texttt{interactsWith}(\texttt{XMMS}, \texttt{Ubuntu})$$
$$\texttt{hasSoftwareVersion}(\texttt{XMMS}, 1.2.10)$$

A Figura A.7 introduz as subclasses de *EndUserApplication*. Todas as subclasses são disjuntas entre si.

- ***DocumentMakingApplication:*** representa a classe dos aplicativos que disponibilizam para seus usuários a edição e formatação de texto permitindo adicionar tabelas, figuras e salvá-lo em um formato web, formato de impressão, entre outro.
  Condições Necessárias:

  (i) **DocumentMakingApplication** é subclasse de **EndUserApplication**.

  $$\texttt{DocumentMakingApplication} \sqsubseteq \texttt{EndUserApplication}$$

  Instanciação: O exemplo demonstra uma possível instância da classe **DocumentMakingApplication**.

  $$\{\ \texttt{LyX}\ \} \sqsubseteq \texttt{DocumentMakingApplication}$$

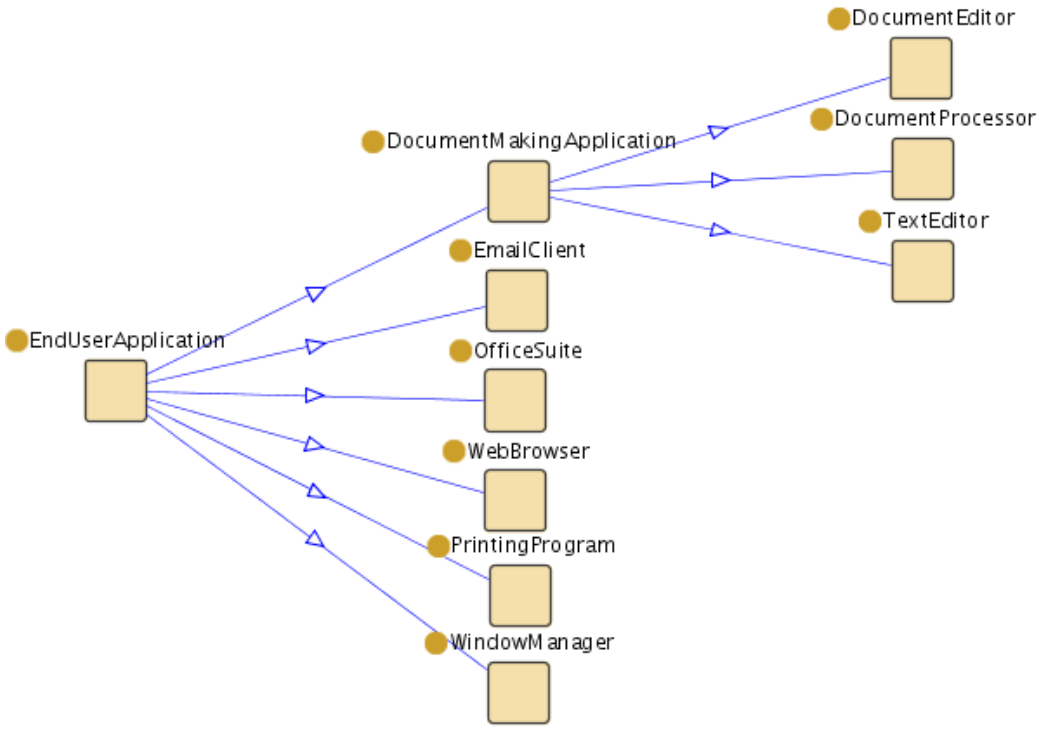

Figura A.7: A classe *EndUserApplication* e suas subclasses

- ***DocumentProcessor:*** esta classe representa o conjunto de programas que validam ou transforma um arquivo texto anotado em um arquivo que pode ser impresso, processado ou simplesmente apresentado na tela.
  Condições Necessárias:

  (i) **DocumentProcessor** é subclasse de **DocumentMakingApplication**.

  $$\texttt{DocumentProcessor} \sqsubseteq \texttt{DocumentMakingApplication}$$

  Instanciação: O exemplo a seguir demonstra uma instância da classe **DocumentProcessor**.

  $$\{\ \texttt{LaTeX}\ \} \sqsubseteq \texttt{DocumentProcessor}$$

- ***TextEditor:*** são programas que permitem visualizar e modificar o conteúdo de arquivos.
  Condições Necessárias:

  (i) **TextEditor** é subclasse de **DocumentMakingApplication**.

  $$\texttt{TextEditor} \sqsubseteq \texttt{DocumentMakingApplication}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **TextEditor**.

  $$\{\ \texttt{emacs}, \texttt{vim}\ \} \sqsubseteq \texttt{TextEditor}$$

- ***EmailClient:*** esta classe representa os programas que podem ser utilizados

por seus usuários para ler, escrever e enviar e-mails.
Condições Necessárias:

(i) **EmailClient** é subclasse de **EndUserApplication**.

$$\texttt{EmailClient} \sqsubseteq \texttt{EndUserApplication}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **EmailClient**.

$$\{\ \texttt{Evolution}, \texttt{Fetchmail}, \texttt{KMail}\ \} \sqsubseteq \texttt{EmailClient}$$

- ***OfficeSuite:*** é um conjunto de programas que são geralmente usados em empresas para escrever documentos e armazená-los, calcular e analisar dados empresariais.
Condições Necessárias:

(i) **OfficeSuite** é subclasse de **EndUserApplication**.

$$\texttt{OfficeSuite} \sqsubseteq \texttt{EndUserApplication}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **OfficeSuite**.

$$\{\ \texttt{MicrosoftOffice}, \texttt{StarOffice}, \texttt{KOffice}\ \} \sqsubseteq \texttt{OfficeSuite}$$

Um **OfficeSuite** pode ser formado por outros aplicativos. A seguir, o exemplo demonstra essa relação para uma das instâncias da classe.

$$\begin{aligned}\texttt{isComposedOf}(\texttt{MicrosoftOffice}, \texttt{Word})\\ \texttt{isComposedOf}(\texttt{MicrosoftOffice}, \texttt{Excel})\\ \texttt{isComposedOf}(\texttt{MicrosoftOffice}, \texttt{PowerPoint})\end{aligned}$$

- ***PrintingProgram:*** é o conjunto dos programas que podem ser usados para imprimir um documento.
Condições Necessárias:

(i) **PrintingProgram** é subclasse de **EndUserApplication**.

$$\texttt{PrintingProgram} \sqsubseteq \texttt{EndUserApplication}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **PrintingProgram**.

$$\{\ \texttt{CUPS}\ \} \sqsubseteq \texttt{PrintingProgram}$$

- ***WebBrowser:*** um *web browser* é uma aplicação que permite aos seus usuários visualizar e interagir com textos, imagens e outras fontes de informação tipicamente localizadas em um *website* na *World Wide Web* ou em uma rede local.
Condições Necessárias:

(i) **WebBrowser** é subclasse de **EndUserApplication**.

$$\texttt{WebBrowser} \sqsubseteq \texttt{EndUserApplication}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **WebBrowser**.

$$\{\ \texttt{InternetExplorer}, \texttt{Firefox}\ \} \sqsubseteq \texttt{WebBrowser}$$

- ***WindowManager:*** é a classe dos programas que controlam a aparencia das janelas em uma interface gráfica. Este tipo de programa provê um padrão de ferramentas e protocolo para a construção de interface gráfica em OpenVMS e sistemas operacionais Unix e derivados.
  Condições Necessárias:

  (i) **WindowManager** é subclasse de **EndUserApplication**.

  $$\texttt{WindowManager} \sqsubseteq \texttt{EndUserApplication}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **WindowManager**.

  $$\{\, \texttt{Metacity}, \texttt{KWin} \,\} \sqsubseteq \texttt{WindowManager}$$

- ***NetworkSoftware:*** esta classe representa softwares que são disponibilizados para seus usuários em um ambiente de rede. São softwares que se destacam por promoverem a colaboração entre seus usários.
  Condições Necessárias:

  (i) **NetworkSoftware** é subclasse de **Software**.

  $$\texttt{WindowManager} \sqsubseteq \texttt{Software}$$

A Figura A.8 introduz as subclasses de *NetworkSoftware*. As subclasses são disjuntas entre si.

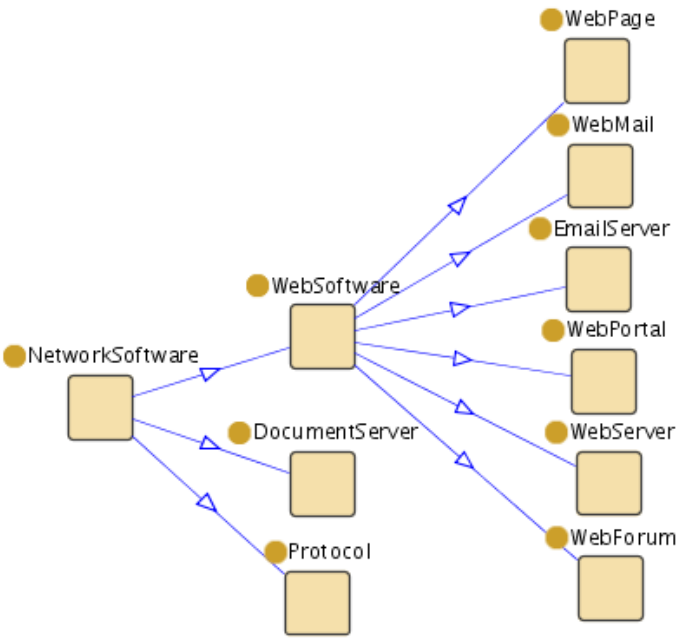

Figura A.8: A classe *NetworkSoftware* e suas subclasses

- ***DocumentServer:*** é a classe dos programas que gerenciam permissões de usuários quanto a manipulação de arquivos contidos no servidor de documentos.
Condições Necessárias:

  (i) **DocumentServer** é subclasse de **EndUserApplication**.

$$\texttt{DocumentServer} \sqsubseteq \texttt{NetworkSoftware}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **DocumentServer**.

$$\{\ \texttt{CERN}, \texttt{Arquemie}\ \} \sqsubseteq \texttt{DocumentServer}$$

É cada vez mais freqüente a utilização de *Document Server* por parte de Instituições academicas. Este tipo de aplicativo é útil para armazenam artigos referentes as pesquisas realizadas dentro dessas instituições de forma que os usuários possam acessar.

- ***Protocol:*** esta classe representa os protocolos que controlam e habilitam conexões, comunicações e transferência de dados.
Condições Necessárias:

  (i) **Protocol** é subclasse de **NetworkSoftware**.

$$\texttt{Protocol} \sqsubseteq \texttt{NetworkSoftware}$$

Instanciação: O exemplo a seguir demontra instâncias da classe **Protocol**.

$$\{\ \texttt{TCP} - \texttt{IP}, \texttt{UDP}, \texttt{IPX}\ \} \sqsubseteq \texttt{Protocol}$$

- ***WebSoftware:*** esta classe representa o conjunto dos softwares implementados para a Web.
Condições Necessárias:

  (i) **WebSoftware** é subclasse de **NetworkSoftware**.

$$\texttt{WebSoftware} \sqsubseteq \texttt{NetworkSoftware}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***WebSoftware****, pode-se dizer:*

  (i) **hasAddress** é uma propriedade de tipo de dados (Equação A.66) cujo domínio é a classe **WebSoftware** (Equação A.67) e o contradomínio é **XMLSchema:string** (Equação A.68).

$$\texttt{hasAddress} \in \texttt{P}_\texttt{D} \tag{A.66}$$

$$\top \sqsubseteq \forall\, \texttt{hasAddress}\ .\ \texttt{WebSoftware}\ \ (\texttt{WebSoftware} \neq \bot) \tag{A.67}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasAddress}^- \ .\ \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \tag{A.68}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **WebSoftware**.

$$\{\ \texttt{GoogleEarth}\ \} \sqsubseteq \texttt{WebSoftware}$$
$$\texttt{hasAddress}(\texttt{GoogleEarth}, \texttt{http} : //\texttt{earth.google.com})$$

A seguir serão detalhada as subclasses de **WebSoftware**. Todas elas são disjuntas entre si.

- ***EmailServer:*** esta classe representa o conjunto de programas que recebem emails transmitidos via rede de computadores e os armazenam até que o destinatário dos emails os solicite via um *email client*.
  Condições Necessárias:

  (i) **EmailServer** é subclasse de **WebSofware**.

  $$\texttt{EmailServer} \sqsubseteq \texttt{WebSoftware}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **EmailServer**.

  $$\{\ \texttt{Merak}, \texttt{Sendmail}, \texttt{Scalix}\ \} \sqsubseteq \texttt{EmailServer}$$

- ***WebForum:*** é um software disponível na World Wide Web cuja finalidade é facilitar discussões e postagem dos usuários sobre diversos assuntos que sejam de interesse de um grupo.
  Condições Necessárias:

  (i) **WebForum** é subclasse de **WebSoftware**.

  $$\texttt{WebForum} \sqsubseteq \texttt{WebSoftware}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **InternetForum**.

  $$\{\ \texttt{LinuxForum}\ \} \sqsubseteq \texttt{InternetForum}$$
  $$\texttt{hasAddress}(\texttt{LinuxForum}, \texttt{www.linuxforums.org})$$

- ***WebMail:*** softwares que permitem a manipulação de emails como pesquisa, escrita, envio e exclusão via Web.
  Condições Necessárias:

  (i) **WebMail** é subclasse de **WebSoftware**.

  $$\texttt{WebMail} \sqsubseteq \texttt{WebSoftware}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **WebMail**.

  $$\{\ \texttt{Gmail}, \texttt{YahooMail}, \texttt{Hotmail}\ \} \sqsubseteq \texttt{WebMail}$$

- ***WebPage:*** é um dos mais simples software para a Web. A principal finalidade de uma página web é a exposição de algum conteúdo informativo e pode ser acessado por um web browser. Esta informação geralmente está no formato HTML ou XHTML e pode ser disponibilizar a navegação para outras páginas web via hiperlinks.
  Condições Necessárias:

  (i) **WebPage** é subclasse de **WebSoftware**.

  $$\texttt{WebPage} \sqsubseteq \texttt{WebSoftware}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **WebPage**.

$$\{ \texttt{BR} - \texttt{Linux} \} \sqsubseteq \texttt{WebPage}$$
$$\texttt{hasAddress}(\texttt{BR} - \texttt{Linux}, \texttt{http} : //\texttt{br} - \texttt{linux.org})$$

- ***WebPortal:*** trata-se de um tipo de software que concentra vários outros softwares web como webmail, forum, webpage, pesquisas sofisticadas entre outros serviços. Geralmente um portal web é referência para a área de conhecimento em que ele se concentra.
  Condições Necessárias:

  (i) **WebPortal** é subclasse de **WebSoftware**.

  $$\texttt{WebPortal} \sqsubseteq \texttt{WebSoftware}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **WebPortal**.

  $$\{ \texttt{SemanticWebCentral} \} \sqsubseteq \texttt{WebPortal}$$
  $$\texttt{hasAddress}(\texttt{SemanticWebCentral}, \texttt{www.semwebcentral.org})$$

- ***WebServer:*** são aplicações que ficam à espera de requisições a recursos web, tais como *web pages*. Após receber as requisições, o *web server* envia os recursos para os usuários de forma que eles possam visualizá-las por seus *browsers* ou utilizá-los via outro aplicativo.
  Condições Necessárias:

  (i) **WebServer** é subclasse de **WebSoftware**.

  $$\texttt{WebServer} \sqsubseteq \texttt{WebSoftware}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **WebServer**.

  $$\{ \texttt{Apache}, \texttt{IIS} \} \sqsubseteq \texttt{WebServer}$$

- ***Ontology:*** esta classe representa os *software* que contêm informações semânticas sobre conceitos de um domínio e as relações entre eles. Com base nessas informações semânticas é possível aplicar um motor de inferência sobre os objetos do domínio e deduzir novas informações além das explícitas que potencializam a normalização do conhecimento comum, o reuso e a representação.
  Condições Necessárias:

  (i) **Ontology** é subclasse de **EndUserApplication**.

  $$\texttt{Ontology} \sqsubseteq \texttt{EndUserApplication}$$

  Disjunções: - *Sobre as classes disjuntas de* ***Ontology****, pode-se dizer:*

  (i) A classe **Ontology** é disjunta das classes **UserInterface**, **SystemSoftware**, **SystemProgram**, **NetworkSoftware**, **EndUserApplication** e **DevelopmentSoftware**

já que estas classes não compartilham instâncias.

$$\texttt{Ontology} \sqsubseteq \begin{pmatrix} \neg\ \texttt{UserInterface} \sqcap \\ \neg\ \texttt{SystemSoftware} \sqcap \\ \neg\ \texttt{SystemProgram} \sqcap \\ \neg\ \texttt{NetworkSoftware} \sqcap \\ \neg\ \texttt{EndUserApplication} \sqcap \\ \neg\ \texttt{DevelopmentSoftware} \end{pmatrix}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe Ontology.

$$\{\ \texttt{OSOnto}, \texttt{SwTO}, \texttt{Neurontology}\ \} \sqsubseteq \texttt{Ontology}$$

- ***SystemProgram:*** são programas básicos necessários à execução do sistema operacional. Enquanto o kernel gerencia recursos de sistema e provê uma forma de outros programas acessarem esses recursos, os *system programs* (ou *basic system programs*) são necessários para inicializar o sistema, disponibilizar login e acesso ao shell, permitir a visualização de processos, entre outras tarefas. Este conjunto de programas contribuem com a usabilidade de um sistema.
  Condições Necessárias:

  (i) **SystemProgram** é subclasse de **Software**.

  $$\texttt{SystemProgram} \sqsubseteq \texttt{Software}$$

  Disjunções: - *Sobre as classes disjuntas de* ***SystemProgram****, pode-se dizer:*

  (i) A classe **SystemProgram** é disjunta das classes **UserInterface**, **SystemSoftware**, **Ontology**, **NetworkSoftware**, *EndUserApplication* e **DevelopmentSoftware** já que estas classes não compartilham instâncias.

  $$\texttt{SystemProgram} \sqsubseteq \begin{pmatrix} \neg\ \texttt{UserInterface} \sqcap \\ \neg\ \texttt{SystemSoftware} \sqcap \\ \neg\ \texttt{Ontology} \sqcap \\ \neg\ \texttt{NetworkSoftware} \sqcap \\ \neg\ \texttt{EndUserApplication} \sqcap \\ \neg\ \texttt{DevelopmentSoftware} \end{pmatrix}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **SystemProgram**.

  $$\{\ \texttt{cat}, \texttt{cd}, \texttt{chmod}, \texttt{chown}, \texttt{less}, \texttt{ls}, \texttt{more}\ \} \sqsubseteq \texttt{SystemProgram}$$

- ***SystemSoftware:*** é a classe de *software* que funciona como uma camada intermediária entre o *hardware* e as aplicações. O primeiro propósito do *System Software* é sempre que possível, isolar as aplicações do contato direto com detalhes complexos dos computadores especialmente memória, processador, e outros recursos do *hardware*. O segundo propósito deste tipo de *software* é assessorar periféricos como impressoras, teclados, monitores, recursos de comunicação, entre outros.
  Condições Necessárias:

  (i) **SystemSoftware** é subclasse de **Software**.

  $$\texttt{SystemSoftware} \sqsubseteq \texttt{Software}$$

(ii) Indivíduos da classe *SystemSoftware* relacionam-se através da propriedade *interactsWith* com pelo menos um indivíduo da classe resultante entre a união de *Hardware* e *EndUserApplication*.

$$\texttt{SystemSoftware} \sqsubseteq \exists\, \texttt{interactsWith}\, (\texttt{Hardware} \sqcup \texttt{EndUserApplication})$$

Disjunções: - *Sobre as classes disjuntas de* ***SystemSoftware****, pode-se dizer:*

(i) A classe **SystemSoftware** é disjunta das classes **UserInterface**, **Ontology**, **SystemProgram**, **NetworkSoftware**, *EndUserApplication* e **DevelopmentSoftware** já que estas classes não compartilham instâncias.

$$\texttt{SystemSoftware} \sqsubseteq \begin{pmatrix} \neg\, \texttt{UserInterface} \sqcap \\ \neg\, \texttt{SystemProgram} \sqcap \\ \neg\, \texttt{Ontology} \sqcap \\ \neg\, \texttt{NetworkSoftware} \sqcap \\ \neg\, \texttt{EndUserApplication} \sqcap \\ \neg\, \texttt{DevelopmentSoftware} \end{pmatrix}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **SystemSoftware**.

$$\{\, \texttt{OpenGL} \,\} \sqsubseteq \texttt{SystemSoftware}$$

A figura A.9 introduz as subclasses de *SystemSoftware* detalhadas a seguir. As subclasses são disjuntas entre si.

- ***BootLoader:*** representa a classe de *software* responsável em auxiliar a inicialização do sistema operacional em um *hardware*. Quando os computadores são ligados, eles não têm o sistema operacional na memória e em geral, o *hardware* sozinho não realiza este tipo de ação. É justamente neste momento que o *boot loader* é acionado. O *boot loader* por sua vez, aciona outros programas de forma que o sistema operacional possa inicializar.
  Condições Necessárias:

  (i) **BootLoader** é subclasse de **SystemSoftware**.

  $$\texttt{BootLoader} \sqsubseteq \texttt{SystemSoftware}$$

  (ii) Indivíduos da classe *BootLoader* relacionam-se através da propriedade *isBootLoaderOf* com pelo menos um indivíduo da classe *OperatingSystem*.

  $$\texttt{BootLoader} \sqsubseteq \exists\, \texttt{isBootLoaderOf}\, .\, \texttt{OperatingSystem}$$

  Propriedades - *Sobre as propriedades que descrevem a classe* ***BootLoader****, pode-se dizer:*

  (i) **isBootLoaderOf** é uma propriedade objeto (Equação A.69) inversa a **hasBootLoader** (Equação A.70), cujo domínio é **BootLoader** e o contradomínio é **Opera-**

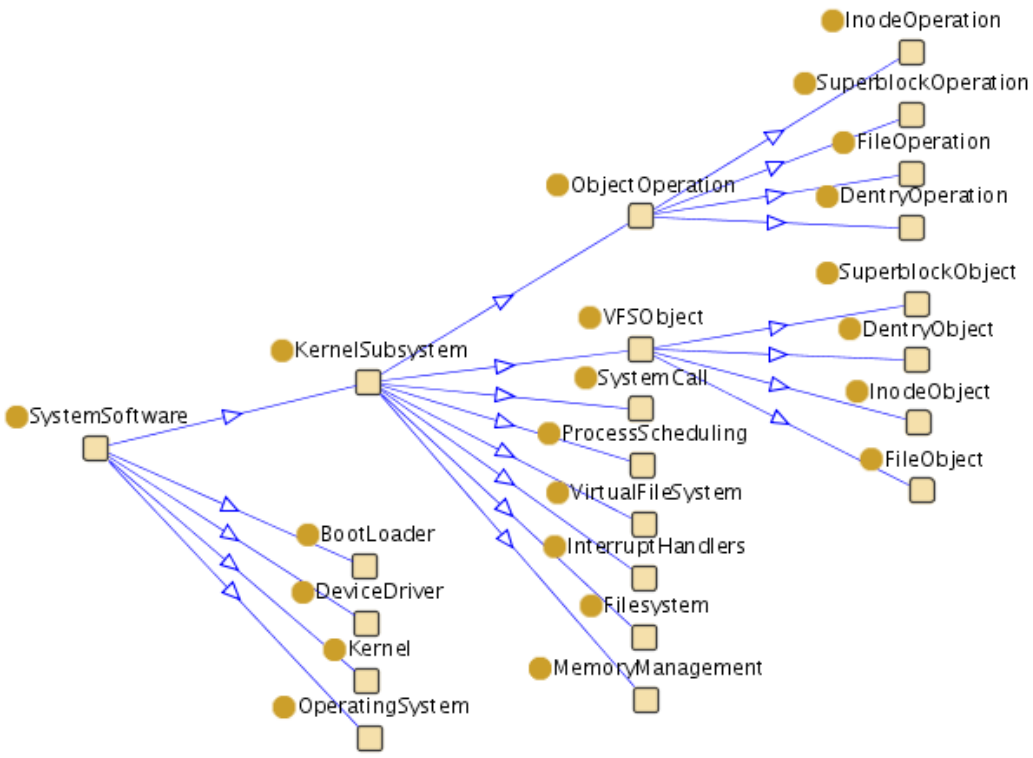

Figura A.9: A classe *System Software* e suas subclasses

tingSystem (Equações A.71 e A.72).

$$\texttt{isBootLoaderOf} \in \texttt{P}_0 \tag{A.69}$$

$$\texttt{isBootLoaderOf} \equiv \texttt{hasBootLoader}^- \tag{A.70}$$

$$\top \sqsubseteq \forall\, \texttt{isBootLoaderOf} \,.\, \texttt{BootLoader} \tag{A.71}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isBootLoaderOf}^- \,. \\ \texttt{OperatingSystem} \end{pmatrix} \tag{A.72}$$

Instanciação: O exemplo a seguir demontra instâncias da classe **BootLoader**.

$$\{\, \texttt{Lilo}, \texttt{GRUB} \,\} \sqsubseteq \texttt{BootLoader}$$

- ***DeviceDriver:*** Um *device driver*, é um tipo de *software* desenvolvido para interagir com recursos do *hardware*. Este tipo de *software* constitui uma interface de comunicação entre barramentos específico do *hardware*. Os drivers são desenvolvidos para atender a um sistema operacional específico.
  Condições Necessárias:

  (i) **DeviceDriver** é subclasse de **SystemSoftware**.

  $$\texttt{DeviceDriver} \sqsubseteq \texttt{SystemSoftware}$$

  (ii) Indivíduos da classe *DeviceDriver* só relacionam-se através da propriedade **isDevice-**

DriverOf) com indivíduos da classe **OperatingSystem**.

$$\texttt{DeviceDriver} \sqsubseteq \forall\, \texttt{isDeviceDriverOf} \,.\, \texttt{OperatingSystem}$$

(iii) Indivíduos da classe *DeviceDriver* relacionam-se através da propriedade **isDeviceDriverOf**) com pelo menos um indivíduo da classe **OperatingSystem**.

$$\texttt{DeviceDriver} \sqsubseteq \exists\, \texttt{isDeviceDriverOf} \,.\, \texttt{OperatingSystem}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***DeviceDriver****, pode-se dizer:*

(i) **isDeviceDriverOf** é uma propriedade objeto (Equação A.73) inversa a **hasDeviceDriver** (Equação A.74), cujo domínio é **DeviceDriver** e o contradomínio é **OperatingSystem** (Equações A.75 e A.76).

$$\texttt{isDeviceDriverOf} \in \texttt{P}_0 \tag{A.73}$$

$$\texttt{isDeviceDriverOf} \equiv \texttt{hasDeviceDriver}^- \tag{A.74}$$

$$\top \sqsubseteq \forall\, \texttt{isDeviceDriverOf} \,.\, \texttt{DeviceDriver} \tag{A.75}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isDeviceDriverOf}^- \,. \\ \texttt{OperatingSystem} \end{pmatrix} \tag{A.76}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **DeviceDriver**.

$$\{\, \texttt{Vidix}, \texttt{SANE} \,\} \sqsubseteq \texttt{DeviceDriver}$$

A seguir é demonstrada a relação de uma das instâncias com a propriedade P_O que descreve a classe.

$$\texttt{isDeviceDriverOf}(\texttt{Vidix}, \texttt{Ubuntu})$$

- ***Kernel:*** trata-se do núcleo de um sistema operacional. O kernel realiza tarefas como gerenciamento de processos, gerenciamento do *hardware*, chamada de sistema, entre outras atividades.
  Condições Necessárias:

  (i) **Kernel** é subclasse de **SystemSoftware**.

  $$\texttt{Kernel} \sqsubseteq \texttt{SystemSoftware}$$

  (ii) Indivíduos da classe *Kernel* só relacinam-se através da propriedade *isKernelOf* com indivíduos da classe *OperatingSystem*.

  $$\texttt{Kernel} \sqsubseteq \forall\, \texttt{isKernelOf} \,.\, \texttt{OperatingSystem}$$

  (iii) Indivíduos da classe *Kernel* relacinam-se através da propriedade *isKernelOf* com pelo menos um indivíduo da classe *OperatingSystem*.

  $$\texttt{Kernel} \sqsubseteq \exists\, \texttt{isKernelOf} \,.\, \texttt{OperatingSystem}$$

  (iv) Indivíduos da classe *Kernel* só relacionam-se através da propriedade *hasSubsystem* com indivíduos da classe *KernelSubsystem*.

  $$\texttt{Kernel} \sqsubseteq \forall\, \texttt{hasSubsystem} \,.\, \texttt{KernelSubsystem}$$

  (v) Indivíduos da classe *Kernel* relacionam-se através da propriedade *hasSubsystem* com

pelo menos um indivíduo da classe *KernelSubsystem*.

$$\texttt{Kernel} \sqsubseteq \exists\, \texttt{hasSubsystem} \,.\, \texttt{KernelSubsystem}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***Kernel****, pode-se dizer:*

(i) **isKernelOf** é uma propriedade objeto (Equação A.77) inversa a **hasKernel** (Equação A.78), cujo domínio é **Kernel** e o contradomínio é **OperatingSystem** (Equações A.79 e A.80).

$$\texttt{isKernelOf} \in \texttt{P}_0 \tag{A.77}$$
$$\texttt{isKernelOf} \equiv \texttt{hasKernel}^- \tag{A.78}$$
$$\top \sqsubseteq \forall\, \texttt{isKernelOf} \,.\, \texttt{Kernel} \tag{A.79}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isKernelOf}^- \,. \\ \texttt{OperatingSystem} \end{pmatrix} \tag{A.80}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **Kernel**.

$$\{\, \texttt{LinuxKernel}, \texttt{UnixKernel}, \texttt{MinixKernel} \,\} \sqsubseteq \texttt{Kernel}$$

A seguir, uma das instâncias da classe é detalhada.

$$\texttt{hasSoftwareVersion}(\texttt{LinuxKernel}, \texttt{2.6.22.1})$$
$$\texttt{isKernelOf}(\texttt{LinuxKernel}, \texttt{Ubuntu})$$
$$\texttt{isKernelOf}(\texttt{LinuxKernel}, \texttt{Debian})$$

- ***KernelSubsystem:*** esta classe representa o conjunto dos subsistemas do kernel.
Condições Necessárias:

(i) **KernelSubsystem** é subclasse de **SystemSoftware**.

$$\texttt{Kernel} \sqsubseteq \texttt{SystemSoftware}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***KernelSubsystem****, pode-se dizer:*

(i) **isSubsystemOf** é uma propriedade objeto (Equação A.81) inversa a **hasSubsystem** (Equação A.82), cujo domínio é **KernelSubsystem** e o contradomínio é **Kernel** (Equações A.83 e A.84).

$$\texttt{isSubsystemOf} \in \texttt{P}_0 \tag{A.81}$$
$$\texttt{isSubsystemOf} \equiv \texttt{hasSubsystem}^- \tag{A.82}$$
$$\top \sqsubseteq \forall\, \texttt{isSubsystemOf} \,.\, \texttt{KernelSubsystem} \tag{A.83}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isSubsystemOf}^- \,. \\ \texttt{Kernel} \end{pmatrix} \tag{A.84}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **KernelSubsystem**.

$$\{\, \texttt{InputOutputScheduler} \,\} \sqsubseteq \texttt{KernelSubsystem}$$

A classe **KernelSubsystem** tem oito subclasses disjuntas entre si que serão

detalhadas a seguir. Como um dos objetivos deste trabalho é representar parte do conhecimento do Linux através da OSOnto, para as próximas classes serão demonstradas instâncias ligadas aos subsistemas do Linux.

- ***Filesystem:*** Esta classe representa o conjunto dos sistemas de arquivos. Eles são responsáveis em organizar, escrever e ler de uma unidade de armazenamento os dados em formato de arquivos. Os sistemas de arquivos são programados para fornecer uma interface de abstração e uma estrutura de dados esperada pelo *Virtual File System* (VFS).
  Condições Necessárias:

  (i) **Filesystem** é subclasse de **KernelSubsystem**.

  $$\texttt{Filesystem} \sqsubseteq \texttt{KernelSubsystem}$$

  (ii) Os sistemas de arquivo possuem uma interface virtual comum que os atende com a finalidade de normalizar os vários tipos de sistema de arquivos. Desta forma, indivíduos da classe *Filesystem* só relacionam-se através da propriedade *hasVirtualInterface* com indivíduos da classe *VirtualFileSystem*.

  $$\texttt{Filesystem} \sqsubseteq \forall\, \texttt{hasVirtualInterface} \,.\, \texttt{VirtualFileSystem}$$

  (iii) Indivíduos da classe *Filesystem* relacionam-se através da propriedade *hasVirtualInterface* com pelo menos um indivíduo da classe *VirtualFileSystem*.

  $$\texttt{Filesystem} \sqsubseteq \exists\, \texttt{hasVirtualInterface} \,.\, \texttt{VirtualFileSystem}$$

  Propriedades - *Sobre as propriedades que descrevem a classe **Kernel**, pode-se dizer:*

  (i) **hasVirtualInterface** é uma propriedade objeto (Equações A.85) inversa a **implementsInterface** (Equação A.86), cujo domínio é **Filesystem** e o contradomínio é **VirtualFileSystem** (Equações A.87 e A.88).

  $$\texttt{hasVirtualInterface} \in \texttt{P}_0 \tag{A.85}$$

  $$\texttt{hasVirtualInterface} \equiv \texttt{implementsInterface}^- \tag{A.86}$$

  $$\top \sqsubseteq \forall\, \texttt{hasVirtualInterface} \,.\, \texttt{FileSystem} \tag{A.87}$$

  $$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasVirtualInterface}^- \,. \\ \texttt{VirtualFileSystem} \end{pmatrix} \tag{A.88}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **Filesystem**. No Linux, são mais de 50 sistemas de arquivos reconhecidos pelo kernel.

  $$\{\, \texttt{ext2}, \texttt{ext3}, \texttt{Ntfs}, \texttt{proc}, \texttt{samba}, \texttt{vfat} \,\} \sqsubseteq \texttt{Filesystem}$$

- ***InterruptHandlers:*** Uma das principais atividades do kernel é o gerenciamento dos *hardwares* conectados ao computador. Uma das formas de realizar esta atividade é por meio de interrupções. Por exemplo, se algo for digitado pelo teclado, o controlador do teclado envia um sinal eletrônico para o processador, que por sua vez alerta o sistema operacional que uma tecla foi pressionada. Os sinais eletrônicos são interrupções. O subsistema do kernel que responde a interrupções específicas é chamado de *interrupt handlers* ou

interrupt service routine (ISR).
Condições Necessárias:

(i) **InterruptHandlers** é subclasse de **KernelSubsystem**.

$$\texttt{InterruptHandlers} \sqsubseteq \texttt{KernelSubsystem}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **InterruptHandlers**.

$$\{\ \texttt{my\_interrupt}\ \} \sqsubseteq \texttt{InterruptHandlers}$$

Na ontologia, *my_interrupt* é uma instância mas em um driver implementado na linguagem C para o Linux, por exemplo, esta interrupção pode ser chamada conforme o trecho de código a seguir:

Este pequeno exemplo ilustra o pontencial de representação da OSOnto. Um simples código

```
1 if (request_irq(irqn, my_interrupt, SA_SHIRQ, "my_device", dev)) {
2         printk(KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
3         return -EIO;
4 }
```

Código A.2: *Codificação em C da instância my_interrupt*

my_interrupt.c pode ser inferido como uma interrupção que pode interagir com um *hardware* específico.

- ***MemoryManagement:*** esta classe representa o conjunto elementos utilizados pelo kernel para o gerenciamento de memória.
  Condições Necessárias:

  (i) **MemoryManagement** é subclasse de **KernelSubsystem**.

  $$\texttt{MemoryManagement} \sqsubseteq \texttt{KernelSubsystem}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **MemoryManagement**.

  $$\{\ \texttt{SlabLayer}, \texttt{Per} - \texttt{CPU}\ \} \sqsubseteq \texttt{MemoryManagement}$$

- ***ObjectOperation:*** esta classe representa o conjunto de operações para cada objeto do sistema de arquivo virtual.
  Condições Necessárias:

  (i) **ObjectOperation** é subclasse de **KernelSubsystem**.

  $$\texttt{ObjectOperation} \sqsubseteq \texttt{KernelSubsystem}$$

(ii) Indivíduos da classe *ObjectOperation* só relacionam-se através da propriedade *isOperationOf* com indivíduos da classe *VFSObject*.

$$\texttt{ObjectOperation} \sqsubseteq \forall\, \texttt{isOperationOf}\,.\,\texttt{VFSObject}$$

(iii) Indivíduos da classe *ObjectOperation* relacionam-se através da propriedade *isOperationOf* com pelo menos um indivíduo da classe *VFSObject*.

$$\texttt{ObjectOperation} \sqsubseteq \exists\, \texttt{isOperationOf}\,.\,\texttt{VFSObject}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***ObjectOperation****, pode-se dizer:*

(i) **isOperationOf** é uma propriedade objeto (Equações A.89) inversa a **hasOperation** (Equação A.90), cujo domínio é **ObjectOperation** e o contradomínio é **VFSObject** (Equações A.91 e A.92).

$$\texttt{isOperationOf} \in \texttt{P}_0 \tag{A.89}$$

$$\texttt{isOperationOf} \equiv \texttt{hasOperation}^- \tag{A.90}$$

$$\top \sqsubseteq \forall\, \texttt{isOperationOf}\,.\,\texttt{ObjectOperation} \tag{A.91}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isOperationOf}^- \,. \\ \texttt{VFSObject} \end{pmatrix} \tag{A.92}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **ObjectOperation**.

$$\{\, \texttt{get\_sb} \,\} \sqsubseteq \texttt{ObjectOperation}$$

No Linux a função get_sb é usada para ler um superblock de uma unidade de armazenamento

e popular o "objeto superblock"criado pelo kernel com o sistema de arquivo adequado.

- ***DentryOperation:*** esta classe representa as operações que podem ser realizadas em um objeto *dentry*.
  Condições Necessárias: **DentryOperation** é subclasse de **ObjectOperation**.

$$\texttt{DentryOperation} \sqsubseteq \texttt{ObjectOperation}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **DentryOperation**.

$$\{\, \texttt{d\_compare} \,\} \sqsubseteq \texttt{DentryOperation}$$

A instância **d_compare** é uma função chamada pelo VFS do Linux para comparar dois arquivos. Vários sistemas de arquivos permitem esta operação no próprio sistema de arquivo virtual. Esta operação pode ser feita pela simples comparação de string. Para alguns sistemas de arquivo como o FAT a comparação por strings não é suficiente porque este tipo de sistema de arquivo não é *case sensitive* e precisa de um método de comparação diferenciado.
A seguir é demonstrada a relação da instância com a propriedade que descreve a classe.

$$\texttt{isOperationOf}(\texttt{d\_compare}, \texttt{d\_op})$$

- ***FileOperation:*** esta classe representa as operações que podem ser realizadas em um objeto *file*.

Condições Necessárias: **FileOperation** é subclasse de **ObjectOperation**.

$$\texttt{FileOperation} \sqsubseteq \texttt{ObjectOperation}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **FileOperation**.

$$\{\ \texttt{mmap}, \texttt{open}\ \} \sqsubseteq \texttt{FileOperation}$$

A instância **mmap** é uma função de memória que mapeia um certo arquivo em um endereço informado. A chamada de sistema mmap() ativa esta função. Já a instância **open** é uma função cria um novo objeto do tipo arquivo e o associa a um objeto **inode**. A chamada de sistema open() ativa esta função.
A seguir é demonstrada a relação das instância com a propriedade que descreve a classe.

$$\texttt{isOperationOf}(\texttt{mmap}, \texttt{f\_op})$$
$$\texttt{isOperationOf}(\texttt{open}, \texttt{f\_op})$$

- ***InodeOperation:*** esta classe representa as operações que podem ser realizadas em um objeto *inode*.
Condições Necessárias: **InodeOperation** é subclasse de **ObjectOperation**.

$$\texttt{InodeOperation} \sqsubseteq \texttt{ObjectOperation}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **InodeOperation**.

$$\{\ \texttt{mmap}, \texttt{open}\ \} \sqsubseteq \texttt{InodeOperation}$$

A instância **mmap** é uma função de memória que mapeia um certo arquivo em um endereço informado. A chamada de sistema mmap() ativa esta função. Já a instância **open** é uma função cria um novo objeto do tipo arquivo e o associa a um objeto **inode**. A chamada de sistema open() ativa esta função.
A seguir é demonstrada a relação das instância com a propriedade que descreve a classe.

$$\texttt{isOperationOf}(\texttt{mmap}, \texttt{f\_op})$$
$$\texttt{isOperationOf}(\texttt{open}, \texttt{f\_op})$$

- ***SuperblockOperation:*** esta classe representa as operações que podem ser realizadas em um objeto *Superblock*.
Condições Necessárias: **SuperblockOperation** é subclasse de **ObjectOperation**.

$$\texttt{SuperblockOperation} \sqsubseteq \texttt{ObjectOperation}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **SuperblockOperation**.

$$\{\ \texttt{put\_super}, \texttt{write\_super}\ \} \sqsubseteq \texttt{SuperblockOperation}$$

A instância **put_super** é uma função chamada pelo VFS na desmontagem para liberar um superblock específico. Já a instância **write_super** é uma função que atualiza o superblock de uma unidade de armazenamento em relação a um superblock especificado. O VFS usa esta função para sincronizar um superblock em memória que foi modificado com o superblock do disco.
A seguir é demonstrada a relação das instância com a propriedade objeto herdada da su-

perclasse.

$$\texttt{isOperationOf}(\texttt{put\_super}, \texttt{s\_op})$$
$$\texttt{isOperationOf}(\texttt{write\_super}, \texttt{s\_op})$$

- ***ProcessScheduling:*** é um subsistema do kernel que seleciona quais são os próximos processos que devem ser executados. O *Process Scheduling* é a base de sistemas operacionais multitarefas.
Condições Necessárias:

  (i) **ProcessScheduling** é subclasse de **KernelSubsystem**.

  $$\texttt{ProcessScheduling} \sqsubseteq \texttt{KernelSubsystem}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **ProcessScheduling**.

  $$\{\ \texttt{PoliceScheduler}, \texttt{O(1)}\ \} \sqsubseteq \texttt{ProcessScheduling}$$

- ***SystemCall:*** funciona como uma interface, um meio comum que leva mensagens entre as aplicações e o kernel.
Condições Necessárias:

  (i) **SystemCall** é subclasse de **KernelSubsystem**.

  $$\texttt{SystemCall} \sqsubseteq \texttt{KernelSubsystem}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **SystemCall**. A chamada sys_silly_copy copia os bytes de um primeiro ponteiro para um segundo ponteiro usando o kernel como intermediário.

  $$\{\ \texttt{sys\_silly\_copy}\ \} \sqsubseteq \texttt{SystemCall}$$

- ***VFSObject:*** esta classe representa os objetos contidos no sistema de arquivo virtual ou a família de estrutura de dados que representa o modelo comum de arquivo. Esta classe contém quatro subclasses que representam os quatro objetos primários.
Condições Necessárias:

  (i) **VFSObject** é subclasse de **KernelSubsystem**.

  $$\texttt{SystemCall} \sqsubseteq \texttt{KernelSubsystem}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **VFSObject**.

  $$\{\ \texttt{file\_system\_type}\ \} \sqsubseteq \texttt{VFSObject}$$

  A instância file_system_type é um objeto, ou estrutura reconhecida pelo sistema de arquivo virtual que descreve o sistema de arquivo e sua capacidade.

- ***DentryObject:*** classe que representa diretórios de um sistema de arquivo virtual. Um diretorio também pode ser um arquivo. Só que para o VFS é util distingüir o que é um diretório já que existem operações específicas que são

realizadas sobre o mesmo.
Condições Necessárias: **DentryObject** é subclasse de **VFSObject**.

$$\texttt{DentryObject} \sqsubseteq \texttt{VFSObject}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **DentryObject**.

$$\{\ \texttt{d\_inode}, \texttt{d\_subdirs}, \texttt{d\_op}\ \} \sqsubseteq \texttt{InodeObject}$$

A instância **d_inode** aponta para os *Inodes*. **d_subdirs** é uma estrutura que registra os subdiretorios e **d_op** é a tabela de operações para o *dentry*.

- ***FileObject:*** usado para representar um arquivo aberto por um processo.
Condições Necessárias: **FileObject** é subclasse de **VFSObject**.

$$\texttt{FileObject} \sqsubseteq \texttt{VFSObject}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **FileObject**.

$$\{\ \texttt{f\_list}, \texttt{f\_op}, \texttt{f\_mode}, \texttt{f\_security}\ \} \sqsubseteq \texttt{FileObject}$$

A instância **f_list** é uma estrutura que contém a lista dos arquivos. **f_op** é um ponteiro para a tabela de operações sobre os arquivos, **f_mode** é uma variável que registra o modo de acesso dos arquivos e **f_security** representa o módulo de segurança.

- ***InodeObject:*** representa todas as informações necessárias para o kernel manipular um arquivo ou diretório.
Condições Necessárias: **InodeObject** é subclasse de **VFSObject**.

$$\texttt{InodeObject} \sqsubseteq \texttt{VFSObject}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **InodeObject**.

$$\{\ \texttt{i\_list}, \texttt{i\_ino}, \texttt{i\_op}, \texttt{i\_mode}\ \} \sqsubseteq \texttt{InodeObject}$$

A instância **i_list** é uma estrutura que lista todos os *Inodes*. **i_ino** é uma variável que registra o *inode number*, **i_op** é a tabela de operações dos inodes e **i_mode** registra as permissões de acesso.

- ***SuperblockObject:*** esta classe representa o objeto *super block* que é implementado por cada sistema de arquivo e é usado para armazenar informações que descrevem um sistema de arquivo específico.
Condições Necessárias: **SuperblockObject** é subclasse de **VFSObject**.

$$\texttt{SuperblockObject} \sqsubseteq \texttt{VFSObject}$$

Indivíduos da classe *SuperblockObject* relacionam-se através da propriedade *mountedFilesys-*

tem com pelo menos um indivíduo da classe *FileSystem*.

$$\texttt{SuperblockObject} \sqsubseteq \exists\, \texttt{mountedFilesystem} \,.\, \texttt{Filesystem}$$

Propriedade - *Sobre a propriedade que descreve a classe* ***SuperblockObject****, pode-se dizer:* **mountedFilesystem** é uma propriedade objeto (Equações A.93), cujo domínio é **SuperblockObject** e o contradomínio é **Filesystem** (Equações A.94 e A.95).

$$\texttt{mountedFilesystem} \in \texttt{P}_0 \tag{A.93}$$

$$\top \sqsubseteq \forall\, \texttt{mountedFilesystem} \,.\, \texttt{SuperblockObject} \tag{A.94}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{mountedFilesystem}^- \,. \\ \texttt{Filesystem} \end{pmatrix} \tag{A.95}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **SuperblockObject**.

$$\{\, \texttt{s\_list}, \texttt{s\_blocksize}, \texttt{s\_op}, \texttt{s\_fs\_info} \,\} \sqsubseteq \texttt{SuperblockObject}$$

A instância **s_list** é uma estrutura que lista todos os *Superblocks*. **s_blocksize** é uma variável que mostra o tamanho do bloco em bytes, **s_op** é a tabela de possíveis operações sobre um superblock e **s_fs_info** é um ponteiro para informações específicas de um sistema de arquivo.

- ***VirtualFileSystem:*** promove uma interface entre o *filesystem* utilizado e as aplicações.
  Condições Necessárias:

  (i) **VirtualFileSystem** é subclasse de **KernelSubsystem**.

  $$\texttt{VirtualFileSystem} \sqsubseteq \texttt{KernelSubsystem}$$

  (ii) Indivíduos da classe *VirtualFileSystem* só relacionam-se através da propriedade *hasObject* com indivíduos da classe *VFSObject*.

  $$\texttt{VirtualFileSystem} \sqsubseteq \forall\, \texttt{hasObject} \,.\, \texttt{VFSObject}$$

  (iii) Todo sistema de arquivo virtual funciona como uma plataforma comum para os vários sistemas de arquivo. Desta forma, indivíduos da classe *VirtualFileSystem* só relacionam-se através da *implementsInterface* com individuos da classe *Filesystem*

  $$\texttt{VirtualFileSystem} \sqsubseteq \forall\, \texttt{implementsInterface} \,.\, \texttt{Filesystem}$$

  (iv) Individuos da classe *VirtualFileSystem* relacionam-se através da propriedade *implementsInterface* com pelo menos um indivíduo da classe *Filesystem*.

  $$\texttt{VirtualFileSystem} \sqsubseteq \exists\, \texttt{implementsInterface} \,.\, \texttt{FileSystem}$$

  Propriedades - *Sobre as propriedades que descrevem a classe* ***VirtualFileSystem****, pode-se dizer:*

  (i) **implementsInterface** é uma propriedade objeto (Equações A.96) inversa a **hasVirtualInterface** (Equação A.97), cujo domínio é **VirtualFileSystem** e o contra-

domínio é **Filesystem** (Equações A.98 e A.99).

$$\texttt{implementsInterface} \in \texttt{P}_0 \tag{A.96}$$
$$\texttt{implementsInterface} \equiv \texttt{hasVirtualInterface}^- \tag{A.97}$$
$$\top \sqsubseteq \forall\, \texttt{implementsInterface} \,.\, \texttt{VirtualFileSystem} \tag{A.98}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{implementsInterface}^- \,. \\ \texttt{Filesystem} \end{pmatrix} \tag{A.99}$$

(ii) **hasObject** é uma propriedade objeto (Equações A.100) inversa a **isObjectOf** (Equação A.101), cujo domínio é **VirtualFileSystem** e o contradomínio é **VFSObject** (Equações A.102 e A.103).

$$\texttt{hasObject} \in \texttt{P}_0 \tag{A.100}$$
$$\texttt{hasObject} \equiv \texttt{isObjectOf}^- \tag{A.101}$$
$$\top \sqsubseteq \forall\, \texttt{hasObject} \,.\, \texttt{VirtualFileSystem} \tag{A.102}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasObject}^- \,. \\ \texttt{VFSObject} \end{pmatrix} \tag{A.103}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **VirtualFilesystem**.

$$\{\, \texttt{vfsmount} \,\} \sqsubseteq \texttt{VirtualFileSystem}$$

A instância demonstra no exemplo, na prática é uma estrutura (*structvfsmount*) usada para descrever a instância montada de um sistema de arquivo usada pelo Linux.

- ***OperatingSystem:*** é um *system software* responsável pela administração do *hardware* e disponibilização de recursos de outros *softwares*. Um sistema operacional inclui dentre vários recursos, o kernel, drivers, boot loader, shell e outras interfaces de usuário.
  Condições Necessárias:

  (i) **OperatingSystem** é subclasse de **SystemSoftware**.

  $$\texttt{OperatingSystem} \sqsubseteq \texttt{SystemSoftware}$$

  (ii) Indivíduos da classe *OperatingSystem* só relacionam-se através da propriedade *hasBootLoader* com indivíduos da classe *BootLoader*.

  $$\texttt{OperatingSystem} \sqsubseteq \forall\, \texttt{hasBootLoader} \,.\, \texttt{BootLoader}$$

  (iii) Indivíduos da classe *OperatingSystem* relacionam-se através da propriedade *hasBootLoader* com pelo menos um indivíduo da classe *BootLoader*.

  $$\texttt{OperatingSystem} \sqsubseteq \exists\, \texttt{hasBootLoader} \,.\, \texttt{BootLoader}$$

  (iv) Indivíduos da classe *OperatingSystem* só relacionam-se através da propriedade *hasDeviceDriver* com indivíduos da classe *DeviceDriver*.

  $$\texttt{OperatingSystem} \sqsubseteq \forall\, \texttt{hasDeviceDriver} \,.\, \texttt{DeviceDriver}$$

  (v) Indivíduos da classe *OperatingSystem* relacionam-se através da propriedade *hasDevi-*

ceDriver com pelo menos um indivíduo da classe *DeviceDriver*.

$$\texttt{OperatingSystem} \sqsubseteq \exists\, \texttt{hasDeviceDriver} \,.\, \texttt{DeviceDriver}$$

(vi) Indivíduos da classe *OperatingSystem* só relacionam-se através da propriedade *hasKernel* com indivíduos da classe *Kernel*.

$$\texttt{OperatingSystem} \sqsubseteq \forall\, \texttt{hasKernel} \,.\, \texttt{Kernel}$$

(vii) Indivíduos da classe *OperatingSystem* relacionam-se através da propriedade *hasKernel* com pelo menos um indivíduo da classe *Kernel*.

$$\texttt{OperatingSystem} \sqsubseteq \exists\, \texttt{hasKernel} \,.\, \texttt{Kernel}$$

(viii) Indivíduos da classe *OperatingSystem* só relacionam-se através da propriedade *hasUserInterface* com indivíduos da classe *UserInterface*.

$$\texttt{OperatingSystem} \sqsubseteq \forall\, \texttt{hasUserInterface} \,.\, \texttt{UserInterface}$$

(ix) Indivíduos da classe *OperatingSystem* relacionam-se através da propriedade *hasUserInterface* com pelo menos um indivíduo da classe *UserInterface*.

$$\texttt{OperatingSystem} \sqsubseteq \exists\, \texttt{hasUserInterface} \,.\, \texttt{UserInterface}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***OperatingSystem****, pode-se dizer:*

(i) **hasBootLoader** é uma propriedade objeto (Equação A.104) inversa a **isBootLoaderOf** (Equação A.105), cujo domínio é **OperatingSystem** e o contradomínio é **BootLoader** (Equações A.106 e A.107).

$$\texttt{hasBootLoader} \in \texttt{P}_0 \tag{A.104}$$

$$\texttt{hasBootLoader} \equiv \texttt{isBootLoader}^- \tag{A.105}$$

$$\top \sqsubseteq \forall\, \texttt{hasBootLoader} \,.\, \texttt{OperatingSystem} \tag{A.106}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasBootLoader}^- \,. \\ \texttt{BootLoader} \end{pmatrix} \tag{A.107}$$

(ii) **hasDeviceDriver** é uma propriedade objeto (Equação A.108) inversa a **isDeviceDriverOf** (Equação A.109), cujo domínio é **OperatingSystem** e o contradomínio é **DeviceDriver** (Equações A.110 e A.111).

$$\texttt{hasDeviceDriver} \in \texttt{P}_0 \tag{A.108}$$

$$\texttt{hasDeviceDriver} \equiv \texttt{isDeviceDriverOf}^- \tag{A.109}$$

$$\top \sqsubseteq \forall\, \texttt{hasDeviceDriver} \,.\, \texttt{OperatingSystem} \tag{A.110}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasDeviceDriver}^- \,. \\ \texttt{DeviceDriver} \end{pmatrix} \tag{A.111}$$

(iii) **hasKernel** é uma propriedade objeto (Equação A.112) inversa a **isKernelOf** (Equação A.113), cujo domínio é **OperatingSystem** e o contradomínio é **Kernel** (Equações

A.114 e A.115).

$$\texttt{hasKernel} \in \texttt{P}_0 \tag{A.112}$$

$$\texttt{hasKernel} \equiv \texttt{isKernelOf}^- \tag{A.113}$$

$$\top \sqsubseteq \forall\, \texttt{hasKernel} \,.\, \texttt{OperatingSystem} \tag{A.114}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasKernel}^- \,. \\ \texttt{Kernel} \end{pmatrix} \tag{A.115}$$

Instanciação: O exemplo a seguir demontra instâncias da classe **OperatingSystem**.

$$\{\, \texttt{Debian}, \texttt{MS} - \texttt{DOS}, \texttt{Windows} \,\} \sqsubseteq \texttt{OperatingSystem}$$

- ***UserInterface:*** trata-se de um tipo de *software* que permite ao usuário interagir com recursos computacionais.
  Condição Necessária e Suficiente:

  (i) Todos os Indivíduos da classe *UserInterface* também são membros de *GraphicalEnvironment* ou da classe *CommandLine*, nunca das duas classes ao mesmo tempo já que elas são disjuntas.

  $$\texttt{UserInterface} \equiv (\texttt{GraphicalEnvironment} \sqcup \texttt{CommandLine})$$

  Condições Necessárias:

  (i) **UserInterface** é subclasse de **Software**.

  $$\texttt{UserInterface} \sqsubseteq \texttt{Software}$$

  (ii) Indivíduos da classe *UserInterface* relacionam-se através da propriedade *isUserInterfaceOf* com pelo menos um indivíduo da classe resultante entre a união de *EndUserApplication* e *SystemSoftware*

  $$\texttt{UserInterface} \sqsubseteq \exists\, \texttt{isUserInterfaceOf}\, (\texttt{EndUserApplication} \sqcup \texttt{OperatingSystem})$$

  Propriedades - *Sobre as propriedades que descrevem a classe* ***UserInterface****, pode-se dizer:*

  (i) **isUserInterfaceOf** é uma propriedade objeto inversa a **hasUserInterface**.

  $$\texttt{isUserInterfaceOf} \in \texttt{P}_0$$
  $$\texttt{isUserInterfaceOf} \equiv \texttt{hasUserInterface}^-$$

  Disjunções: - *Sobre as classes disjuntas de* ***UserInterface****, pode-se dizer:*

  (i) A classe **UserInterface** é disjunta das classes **Ontology**, **SystemSoftware**, **SystemProgram**, **NetworkSoftware**, *EndUserApplication* e **DevelopmentSoftware** já que estas classes não compartilham instâncias.

  $$\texttt{UserInterface} \sqsubseteq \begin{pmatrix} \neg\, \texttt{Ontology} \sqcap \\ \neg\, \texttt{SystemSoftware} \sqcap \\ \neg\, \texttt{SystemProgram} \sqcap \\ \neg\, \texttt{NetworkSoftware} \sqcap \\ \neg\, \texttt{EndUserApplication} \sqcap \\ \neg\, \texttt{DevelopmentSoftware} \end{pmatrix}$$

A seguir serão apresentadas as duas subclasses de UserInterface. Ambas são disjuntas entre si.

- ***CommandLine:*** é um tipo de interface pouco intuitiva, exigindo do usuário um bom domínio dos comando necessários para interagir através da linha de comando.
  Condições Necessárias:

  (i) **CommandLine** é subclasse de **UserInterface**.

  $$\texttt{CommandLine} \sqsubseteq \texttt{UserInterface}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **CommandLine**.

  $$\{\ \texttt{bash}, \texttt{csh}, \texttt{ksh}, \texttt{sh}, \texttt{tcsh}, \texttt{zsh}\ \} \sqsubseteq \texttt{CommandLine}$$

  A instância **bash** também é conhecida como *Bourne Again Shell*. A instância **csh** também é conhecida como *C Shell*, ksh vem de *Korn Shell*, **sh** vem de *Bourne Shell*, a interface **tcsh** é conhecida como *Enhanced C Shell*, e **zsh** vem de *Z Shell*.

- ***GraphicalEnvironment:*** é um tipo de interface freqüentemente referenciada como amigável pois os usuários não precisam se preocupar, a princípio, em aprender comandos para interagir. Também é uma interface reconhecida como intuitiva devido os recurso visuais que auxiliam e conduzem o usuário.
  Condições Necessárias:

  (i) **GraphicalEnvironment** é subclasse de **UserInterface**.

  $$\texttt{GraphicalEnvironment} \sqsubseteq \texttt{UserInterface}$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **GraphicalEnvironment**.

  $$\{\ \texttt{Kde}, \texttt{Gnome}\ \} \sqsubseteq \texttt{GraphicalEnvironment}$$

- ***SoftwareCategory:*** trata-se das categorias que são atribuídas ao software de acordo com o seu tipo de licença.
  Condições Necessárias:

  (i) **SoftwareCategory** é subclasse de **OperatingSystemDomainConcept** já que assim como qualquer software, o sistema operacional se enquadra em uma categoria.

  $$\texttt{SoftwareCategory} \sqsubseteq \texttt{OperatingSystemDomainConcept}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **SoftwareCategory**.

$$\left(\begin{array}{c}\texttt{CommertialSoftware},\\ \texttt{FreeSoftware},\\ \texttt{Freeware},\\ \texttt{OpenSource},\\ \texttt{PrivateSoftware},\\ \texttt{ProprietarySoftware},\\ \texttt{PublicDomain},\\ \texttt{Shareware}\end{array}\right) \sqsubseteq \texttt{SoftwareCategory}$$

Sendo que entre algumas instâncias existe as seguintes particularidades:

$$\begin{aligned}\texttt{owl} : \texttt{differenteFrom}(\texttt{CommertialSoftware}, \texttt{ProprietarySoftware})\\ \texttt{owl} : \texttt{differenteFrom}(\texttt{FreeSoftware}, \texttt{Shareware})\\ \texttt{owl} : \texttt{differenteFrom}(\texttt{FreeSoftware}, \texttt{ProprietarySoftware})\\ \texttt{owl} : \texttt{differenteFrom}(\texttt{FreeSoftware}, \texttt{Freeware})\\ \texttt{owl} : \texttt{differenteFrom}(\texttt{Shareware}, \texttt{PublicDomain})\\ \texttt{owl} : \texttt{differenteFrom}(\texttt{Shareware}, \texttt{OpenSource})\\ \texttt{owl} : \texttt{differenteFrom}(\texttt{Shareware}, \texttt{Freeware})\\ \texttt{owl} : \texttt{differenteFrom}(\texttt{Shareware}, \texttt{FreeSoftware})\end{aligned}$$

A figura A.10, extraída do ambiente Protégé, ilustra a instância *FreeSoftware* e a propriedade *owl:differentFrom*.

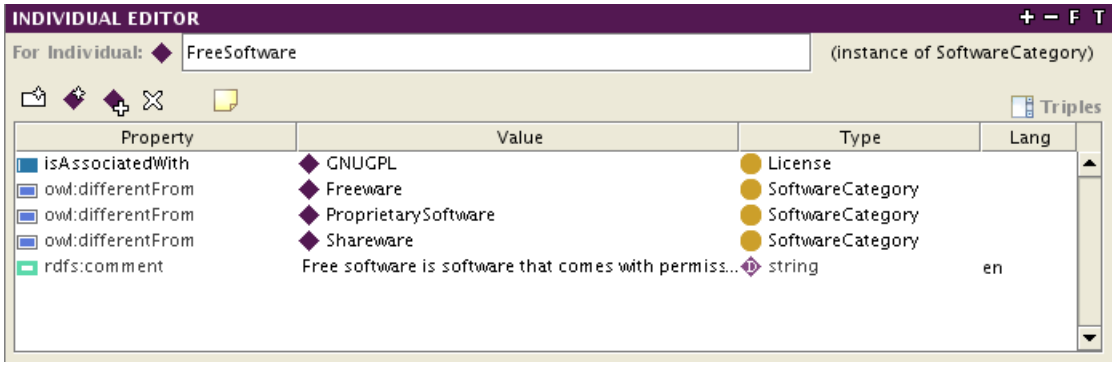

Figura A.10: A instância *FreeSoftware* e sua propriedades

- ***SoftwareDocumentation:*** esta classe concentra as documentações de um software.
  Condições Necessárias:

  (i) **SoftwareDocumentation** é subclasse de **OperatingSystemDomainConcept** já que assim como qualquer software, o sistema operacional tem documentação.

  $$\texttt{SoftwareDocumentation} \sqsubseteq \texttt{OperatingSystemDomainConcept}$$

(ii) Indivíduos da classe *SoftwareDocumentation* só relacionam-se através da propriedade *documents* com indivíduos da classe *Software*.

$$\texttt{SoftwareDocumentation} \sqsubseteq \forall\, \texttt{documents} \,.\, \texttt{Software}$$

(iii) Indivíduos da classe *SoftwareDocumentation* relacionam-se através da propriedade *documents* com pelo menos um indivíduo da classe *Software*.

$$\texttt{SoftwareDocumentation} \sqsubseteq \exists\, \texttt{documents} \,.\, \texttt{Software}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***SoftwareDocumentation****, pode-se dizer:*

(i) **documents** é uma propriedade objeto (Equação A.116), cujo domínio é a classe **SoftwareDocumentation** e o contradomínio é **Software** (Equações A.117 e A.118).

$$\texttt{documents} \in \texttt{P}_\texttt{O} \tag{A.116}$$
$$\top \sqsubseteq \forall\, \texttt{documents} \,.\, \texttt{SoftwareDocumentation} \tag{A.117}$$
$$\top \sqsubseteq \forall\, \texttt{documents}^{-} \,.\, \texttt{Software} \tag{A.118}$$

(ii) Cada documentação dispõe de uma versão. Esta versão pode auxiliar no controle da documentação do software. **hasDocumentationVersion** é uma subpropriedade de tipo de dados (Equações A.119 e A.120) funcional (Equação A.121). Para esta propriedade, o domínio especificado é a classe **SoftwareDocumentation** (Equação A.122) e o contradomínio é **XMLSchema:string** (Equação A.123).

$$\texttt{hasDocumentationVersion} \in \texttt{P}_\texttt{D} \tag{A.119}$$
$$\texttt{hasDocumentationVersion} \sqsubseteq \texttt{version} \tag{A.120}$$
$$\top \sqsubseteq (\leq \texttt{hasDocumentationVersion}\,) \tag{A.121}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasDocumentationVersion} \,.\, \texttt{SoftwareDocumentation} \\ (\texttt{SoftwareDocumentation} \neq \bot) \end{pmatrix} \tag{A.122}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasDocumentationVersion}^{-} \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \tag{A.123}$$

A seguir as subclasses de **SoftwareDocumentation** serão detalhadas a seguir. A Figura A.11 introduz as subclasses.

A classe *MarketingDocumentation* representa o conjunto das documentações de marketing relacionadas a um software. A classe *TechnicalDocumentation* representa o conjunto das documentações técnicas de um software tais como documentações relacionadas ao código-fonte do software, a algoritmos específicos utilizados no software, a interfaces e API's. A classe *UserDocumentation* representa o conjunto das documentações voltadas para o usuário tais como manuais para usuários finais e para administradores do sistema. A classe *UserDocumentation* possui a subclasse *ManPage*. Alguns sistemas como o Unix, Linux e derivados

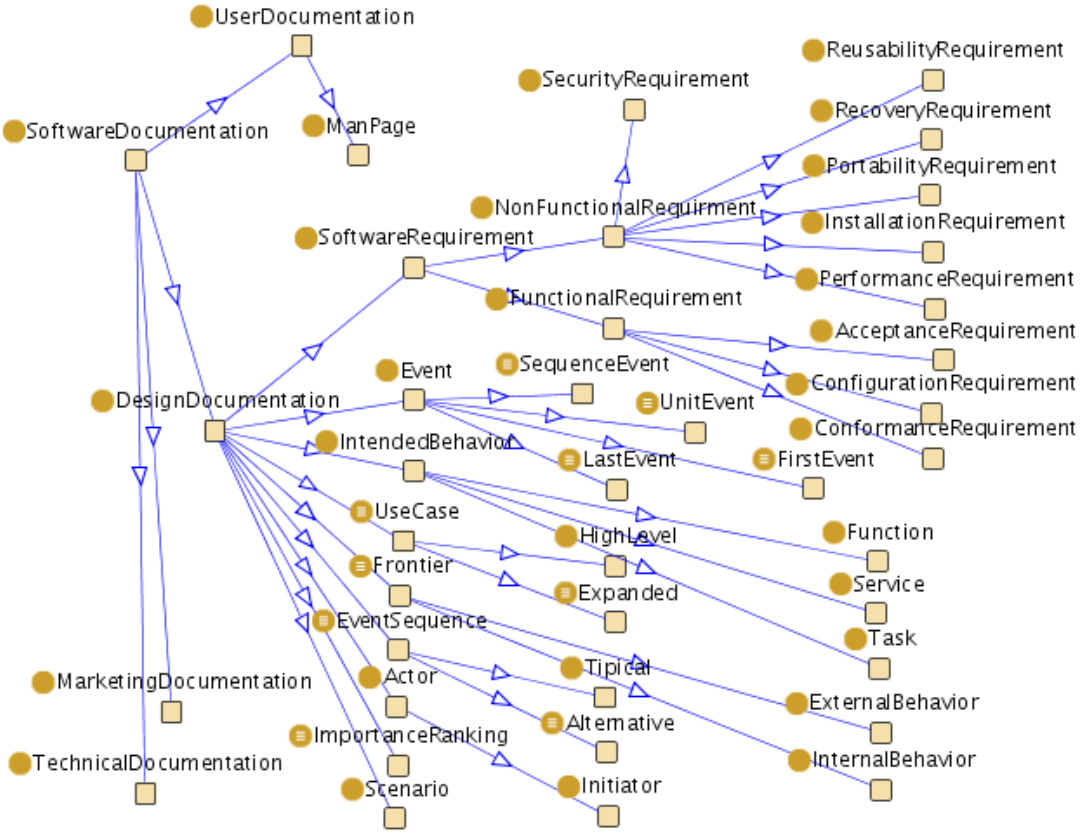

Figura A.11: A classe *SoftwareDocumentation* e suas subclasses

contém uma extensa documentação conhecida como *man pages* (redução de *manual pages*) disponível para seus usuário e a classe *ManPage* representa este tipo de documentação. A seguir a classe *DesignDocumentation*, mais rica em descrições, será discutida.

- ***DesignDocumentation:*** são documentações que fornecem uma visão geral do software. Definem relações do software com o seu ambiente utilização bem como a definição dos princípios de construção que devem ser utilizados no software.
  Condições Necessárias:

  (i) **DesignDocumentation** é subclasse de **SoftwareDocumentation**.

  $$\texttt{DesignDocumentation} \sqsubseteq \texttt{SoftwareDocumentation}$$

  Disjunções: - *Sobre as classes disjuntas de* ***DesignDocumentation****, pode-se dizer:*

  (i) A classe **DesignDocumentation** é disjunta das classes **MarketingDocumentation**, **TechnicalDocumentation** e **UserDocumentation** já que estas classes não

compartilham instâncias.

$$\texttt{DesignDocumentation} \sqsubseteq \begin{pmatrix} \neg\ \texttt{MarketingDocumentation}, \\ \neg\ \texttt{TechnicalDocumentation}, \\ \neg\ \texttt{UserDocumentation} \end{pmatrix}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **DesignDocumentation**.

$$\{\ \texttt{ClassDiagramsTeSG}, \texttt{SequenceDiagramsTeSG}\ \} \sqsubseteq \texttt{DesignDocumentation}$$

A seguir serão detalhadas as subclasses de **DesignDocumentation**. A Figura A.12 introduz as subclasses. Para exemplificar uma possível docu-

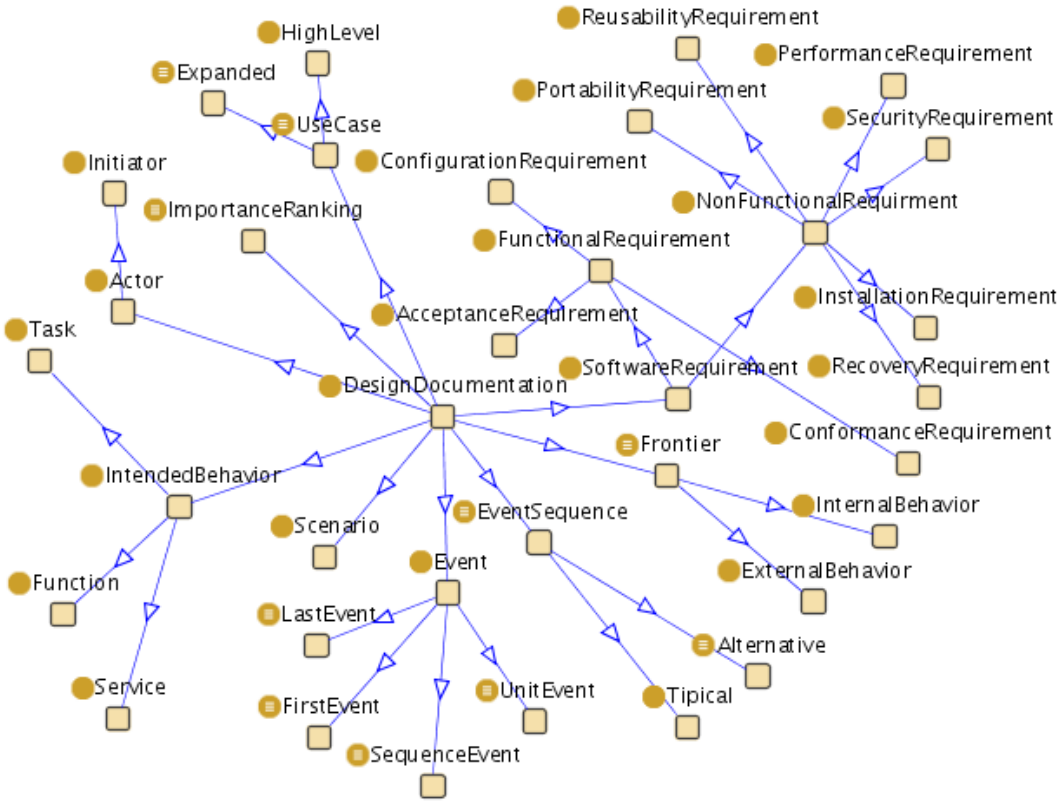

Figura A.12: A classe *DesignDocumentation* e suas subclasses

mentação de um software, escolhemos mais uma vez o Linux por razões já explicadas anteriormente.

Sabemos que no modelo de desenvolvimento do Linux há muito mais esforço concentrado na implementação do que na elaboração de uma documentação

diversificada.

Boa parte da documentação está presente no próprio código-fonte e isso demanda uma certa habilidade do desenvolvedor na hora de consultá-la e atualizá-la. É assim com a documentação dos casos de testes do Linux presente no LTP. Para o desenvolvedor encontrar o código (do caso de teste) desejado, é necessário navegar pela árvore de diretórios, e caso o desenvolvedor não esteja habituado com o conteúdo dos arquivos, será necessário abrí-lo e ler a documentação contida no cabeçalho do arquivo. Caso o arquivo não tenha este resumo, disposto geralmente no cabeçalho do arquivo ou disperso ao longo do código, o desenvolvedor terá que ler o código para então saber exatamente o propósito daquele algoritmo.

Oferecer informação e conhecimento organizado para esses desenvolvedores de forma prática com uma consulta ágil tende a pontecializar o trabalho dos mesmos bem como a qualidade do software. Para isso, foram analisados alguns casos de teste implementados para o **FileSystem** do Linux, presentes no LTP. Foi feito um procedimento de engenharia reversa e do código, foram especificados casos de uso que por sua vez estão presentes nesta ontologia como instâncias. A seguir, as instâncias contidas nas subclasses de **SoftwareDocumentation**, demonstram que é possível organizar e estruturar semanticamente uma informação de origem meramente sintática e pouca organização que depende da interpretação individual de cada pessoa sujeita à falhas.

Agora, de posse desta representação do conhecimento extraída do repositório do LTP é possível criar e inovar através de diversas ferramentas. A TeSG, discutida no capítulo 6, é um exemplo de ferramenta que faz uso desta representação.

- ***Actor:*** é basicamente uma entidade externa (como pessoas, *hardware* ou um *software*) ao sistema que interage com o mesmo. Um ator faz parte de um caso de uso que é um *Design Documentation*.

Condições Necessárias:

(i) **Actor** é subclasse de **DesignDocumentation**.

$$\texttt{Actor} \sqsubseteq \texttt{DesignDocumentation}$$

(ii) A classe *Actor* é subclasse da classe união anônima entre *Hardware*, *Software* e *Person*.

$$\texttt{Actor} \equiv \texttt{Person} \sqcup \texttt{Hardware} \sqcup \texttt{Software}$$

(iii) Indivíduos da classe *Actor* só relacionam-se através da propriedade *isActorOf* com indivíduos da classe *UseCase*.

$$\texttt{Actor} \sqsubseteq \forall\, \texttt{isActorOf} \,.\, \texttt{UseCase}$$

(iv) Indivíduos da classe *Actor* relacionam-se através da propriedade *isActorOf* com pelo menos um indivíduo da classe *UseCase*.

$$\texttt{Actor} \sqsubseteq \exists\, \texttt{isActorOf} \,.\, \texttt{UseCase}$$

(v) Uma fronteira, no contexto dos casos de uso, delimita o comportamento interno e externo ao sistema. Indivíduos da classe *Actor* só relacionam-se através da propriedade *isDelimitedBy* com indivíduos da classe *ExternalBehavior*.

$$\texttt{Actor} \sqsubseteq \forall\, \texttt{isDelimitedBy} \,.\, \texttt{ExternalBehavior}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***Actor****, pode-se dizer:*

(i) **isActorOf** é uma propriedade objeto inversa a **hasActor**, cujo domínio é **Actor** e o contradomínio é **UseCase**.

$$\texttt{isActorOf} \in \texttt{P}_0 \tag{A.124}$$
$$\texttt{isActorOf} \sqsubseteq \texttt{hasActor} \tag{A.125}$$
$$\top \sqsubseteq \forall\, \texttt{isActorOf} \,.\, \texttt{Actor} \tag{A.126}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isActorOf}^- \,. \\ \texttt{UseCase} \end{pmatrix} \tag{A.127}$$

(ii) **isDelimitedBy** é uma propriedade objeto inversa a **delimits**, cujo domínio é **Actor** e o contradomínio é **ExternalBehavior**.

$$\texttt{isDelimitedBy} \in \texttt{P}_0 \tag{A.128}$$
$$\texttt{isDelimitedBy} \sqsubseteq \texttt{delimits} \tag{A.129}$$
$$\top \sqsubseteq \forall\, \texttt{isDelimitedBy} \,.\, \texttt{Actor} \tag{A.130}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isDelimitedBy}^- \,. \\ \texttt{ExternalBehavior} \end{pmatrix} \tag{A.131}$$

(iii) **hasActorDescription** é uma propriedade de tipo de dados (Equação A.132) funcional (Equação A.133). Para esta propriedade, o domínio especificado é a classe **Actor**

(Equação A.134) e o contradomínio é **XMLSchema:string** (Equação A.135).

$$\texttt{hasActorDescription} \in \texttt{P}_\texttt{D} \tag{A.132}$$
$$\top \sqsubseteq (\leq\ 1\ \texttt{hasActorDescription}) \tag{A.133}$$
$$\top \sqsubseteq \forall\, \texttt{hasActorDescription}\,.\,\texttt{Actor}\ (\texttt{Actor} \neq \bot) \tag{A.134}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasActorDescription}^{-}\,.\,\texttt{XMLSchema:string} \\ (\texttt{XMLSchema:string} \neq \bot) \end{pmatrix} \tag{A.135}$$

(iv) **hasActorName** é uma propriedade de tipo de dados (Equação A.136) funcional (Equação A.137). Para esta propriedade, o domínio especificado é a classe **Actor** (Equação A.138) e o contradomínio é **XMLSchema:string** (Equação A.139).

$$\texttt{hasActorName} \in \texttt{P}_\texttt{D} \tag{A.136}$$
$$\top \sqsubseteq (\leq\ 1\ \texttt{hasActorName}) \tag{A.137}$$
$$\top \sqsubseteq \forall\, \texttt{hasActorName}\,.\,\texttt{Actor}\ (\texttt{Actor} \neq \bot) \tag{A.138}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasActorName}^{-}\,.\,\texttt{XMLSchema:string} \\ (\texttt{XMLSchema:string} \neq \bot) \end{pmatrix} \tag{A.139}$$

Disjunções: - *Sobre as classes disjuntas de **Actor**, pode-se dizer:*

(i) A classe **Actor** é disjunta das classes **SoftwareRequirement**, **Event**, **EventSequence**, **ImportanceRanking**, **UseCase** e **InternalBehavior** já que estas classes não compartilham instâncias.

$$\texttt{Actor} \sqsubseteq \begin{pmatrix} \neg\ \texttt{SoftwareRequirement}, \\ \neg\ \texttt{Event}, \\ \neg\ \texttt{EventSequence}, \\ \neg\ \texttt{ImportanceRanking}, \\ \neg\ \texttt{UseCase}, \\ \neg\ \texttt{InternalBehavior}, \end{pmatrix}$$

- ***Initiator:*** é um tipo de ator que inicializa um caso de uso.
Condições Necessárias:

(i) **Initiator** é subclasse de **Actor**.

$$\texttt{Initiator} \sqsubseteq \texttt{Actor}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **Initiator**.

$$\{\ \texttt{Initiator\_1}\ \} \sqsubseteq \texttt{InitiatorhasActorName}(\texttt{Initiator\_1}, \texttt{Afonso})$$

Este ator inicializa os seguintes casos de uso:

$$\texttt{startAn}\ \left(\begin{array}{ll}\texttt{Initiator\_1}, & \left(\begin{array}{c}\texttt{RemoveNotEmptyDirectory}\\ \texttt{MountPartition}\\ \texttt{CreateDirectory}\\ \texttt{RemoveSymbolicLink}\\ \texttt{CangePermission}\\ \texttt{CreateEmptyFile}\\ \texttt{RemoveFile}\\ \texttt{RemoveEmptyDirectory}\\ \texttt{CreateSymbolicLink}\\ \texttt{OpenFile}\\ \texttt{CloseFile}\\ \texttt{UmountPartition}\\ \texttt{MoveFileOrDirectory}\\ \texttt{CopyFile}\\ \texttt{ReadFile}\\ \texttt{WriteFile}\\ \texttt{CopyDirectory}\end{array}\right)\end{array}\right)$$

- ***Event:*** um evento é um conjunto de atividades que são invocadas por estímulos de um ator.
  Condições Necessárias:

  (i) **Event** é subclasse de **DesignDocumentation**.

  $$\texttt{Event} \sqsubseteq \texttt{DesignDocumentation}$$

  (ii) Os eventos podem estar associados e seguir uma seqüência. Desta forma, indivíduos da classe *Event* só relacionam-se através da propriedade *isEventOf* com indivíduos da classe *EventSequence*

  $$\texttt{Event} \sqsubseteq \forall\ \texttt{isEventOf}\ .\ \texttt{EventSequence}$$

  Propriedades - *Sobre as propriedades que descrevem a classe* ***Event****, pode-se dizer:*

  (i) **isEventOf** é uma propriedade objeto inversa a **hasEvent**, cujo domínio é **Event** e o contradomínio é **EventSequence**.

  $$\texttt{isActorOf} \in \texttt{P}_0 \tag{A.140}$$
  $$\texttt{isActorOf} \sqsubseteq \texttt{hasActor}\top \sqsubseteq \forall\ \texttt{isActorOf}\ .\ \texttt{Actor} \tag{A.141}$$
  $$\top \sqsubseteq \forall \begin{pmatrix}\texttt{isActorOf}^{-}\ .\\ \texttt{UseCase}\end{pmatrix} \tag{A.142}$$

  (ii) **isDelimitedBy** é uma propriedade objeto inversa a **delimits**, cujo domínio é **Actor** e o contradomínio é **ExternalBehavior**.

  $$\texttt{isDelimitedBy} \in \texttt{P}_0 \tag{A.143}$$
  $$\texttt{isDelimitedBy} \sqsubseteq \texttt{delimits} \tag{A.144}$$
  $$\top \sqsubseteq \forall\ \texttt{isDelimitedBy}\ .\ \texttt{Actor} \tag{A.145}$$
  $$\top \sqsubseteq \forall \begin{pmatrix}\texttt{isDelimitedBy}^{-}\ .\\ \texttt{ExternalBehavior}\end{pmatrix} \tag{A.146}$$

(iii) **hasEventDescription** é uma propriedade de tipo de dados (Equação A.3) funcional (Equação A.147). Para esta propriedade, o domínio especificado é a classe **Event** (Equação A.149) e o contradomínio é **XMLSchema:string** (Equação A.150).

$$\texttt{hasEventDescription} \in \texttt{P}_\texttt{D} \tag{A.147}$$
$$\top \sqsubseteq (\leq\ 1\ \texttt{hasEventDescription}) \tag{A.148}$$
$$\top \sqsubseteq \forall\, \texttt{hasEventDescription}\,.\,\texttt{Event}\ (\texttt{Event} \neq \bot) \tag{A.149}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasEventDescription}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \tag{A.150}$$

(iv) **hasEventIDNumber** é uma propriedade de tipo de dados (Equação A.151) funcional (Equação A.152). Para esta propriedade, o domínio especificado é a classe **Event** (Equação A.153) e o contradomínio é **XMLSchema:int** (Equação A.154).

$$\texttt{hasEventIDNumber} \in \texttt{P}_\texttt{D} \tag{A.151}$$
$$\top \sqsubseteq (\leq\ 1\ \texttt{hasEventIDNumber}) \tag{A.152}$$
$$\top \sqsubseteq \forall\, \texttt{hasEventIDNumber}\,.\,\texttt{Event}\ (\texttt{Event} \neq \bot) \tag{A.153}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasEventIDNumber}^- \,.\, \texttt{XMLSchema} : \texttt{int} \\ (\texttt{XMLSchema} : \texttt{int} \neq \bot) \end{pmatrix} \tag{A.154}$$

(v) **hasEventPriority** é uma propriedade de tipo de dados (Equação A.155) funcional (Equação A.156). Para esta propriedade, o domínio especificado é a classe **Event** (Equação A.157) e o contradomínio é **XMLSchema:int** (Equação A.158).

$$\texttt{hasEventPriority} \in \texttt{P}_\texttt{D} \tag{A.155}$$
$$\top \sqsubseteq (\leq\ 1\ \texttt{hasEventPriority}) \tag{A.156}$$
$$\top \sqsubseteq \forall\, \texttt{hasEventPriority}\,.\,\texttt{Event}\ (\texttt{Event} \neq \bot) \tag{A.157}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasEventPriority}^- \,.\, \texttt{XMLSchema} : \texttt{int} \\ (\texttt{XMLSchema} : \texttt{int} \neq \bot) \end{pmatrix} \tag{A.158}$$

Essa propriedade tem alguns valores inteiros definidos pelo construtor owl:oneOf onde um evento pode ter prioridade de 1 a 5, onde 1 significa baixa prioridade e 5 alta prioridade.

$$\{\ 1, 2, 3, 4, 5\ \} \sqsubseteq \texttt{hasEventPriority}$$

Disjunções: - *Sobre as classes disjuntas de* ***Event****, pode-se dizer:*

(i) A classe **Event** é disjunta das classes **ExternalBehavior**, **SoftwareRequirement**, **UseCase**, **Actor**, **ImportanceRanking** e **Frontier** já que estas classes não compartilham instâncias.

$$\texttt{Event} \sqsubseteq \begin{pmatrix} \neg\ \texttt{ExternalBehavior}, \\ \neg\ \texttt{SoftwareRequirement}, \\ \neg\ \texttt{UseCase}, \\ \neg\ \texttt{Actor}, \\ \neg\ \texttt{ImportanceRanking}, \\ \neg\ \texttt{Frontier}, \end{pmatrix}$$

A seguir serão detalhadas as quatro subclasses de *Event*. Elas não são disjuntas entre si já que podem compartilhar instâncias.

- ***FirstEvent:*** esta classe representa o conjunto dos primeiros eventos de uma seqüência.
  Condição Necessária e Suficiente:

  (i) Um indivíduo pode ser inferido como *FirsEvent* se e somente se ele não tiver um antecessor e tiver exatamente um sucessor.

$$\texttt{FirstEvent} \equiv (\texttt{hasSubsequent} = 1 \,.\, \texttt{Event})$$

$$\texttt{FirstEvent} \equiv \neg\ (\ \texttt{hasPrevious} \,.\, \texttt{Event})$$

  Condições Necessárias:

  (i) **FirstEvent** é subclasse de **Event**.

$$\texttt{FirstEvent} \sqsubseteq \texttt{Event}$$

  Propriedades - *Sobre as propriedades que descrevem a classe* ***FirstEvent****, pode-se dizer:*

  (i) **hasSubsequent** é uma propriedade objeto funcional.

$$\texttt{hasSubsequent} \in \texttt{P}_0$$
$$\top \sqsubseteq (\leq\ 1\ \texttt{hasSubsequent})$$

  (ii) **hasPrevious** é uma propriedade objeto funcional.

$$\texttt{hasPrevious} \in \texttt{P}_0$$
$$\top \sqsubseteq (\leq\ 1\ \texttt{hasPrevious})$$

  Instanciação: O exemplo a seguir demonstra instâncias da classe **FirstEvent**.

$$\{\ \texttt{FirstEvent\_1}\ \} \sqsubseteq \texttt{FirstEvent}$$

  A Figura A.13 mostra a relação da classe **FirstEvent** com suas instâncias. Este evento tem a seguinte descrição:

$$\texttt{hasEventDescription}\left(\texttt{FirstEvent\_1}, \left(\begin{array}{c}\text{Usuário usa o progrma touch}\\ \text{passando como argumento o nome}\\ \text{do arquivo ou o caminho absoluto}\\ \text{ou relativo no qual ele deseja}\\ \text{criar o arquivo}\end{array}\right)\right)$$

  Este evento pertence a seguinte seqüência:

$$\texttt{isEventOf}(\texttt{FirstEvent\_1}, \texttt{Tipical\_1})$$

- ***LastEvent:*** trata-se do último evento em uma seqüência.
  Condição Necessária e Suficiente:

  (i) Um indivíduo pode ser inferido como *LastEvent* se e somente se ele tiver extamente um antecessor e não tiver um sucessor.

$$\texttt{LastEvent} \equiv (\texttt{hasPrevious} = 1\ \texttt{Event})$$

$$\texttt{LastEvent} \equiv \neg\ (\texttt{hasSubsequent} \,.\, \texttt{Event})$$

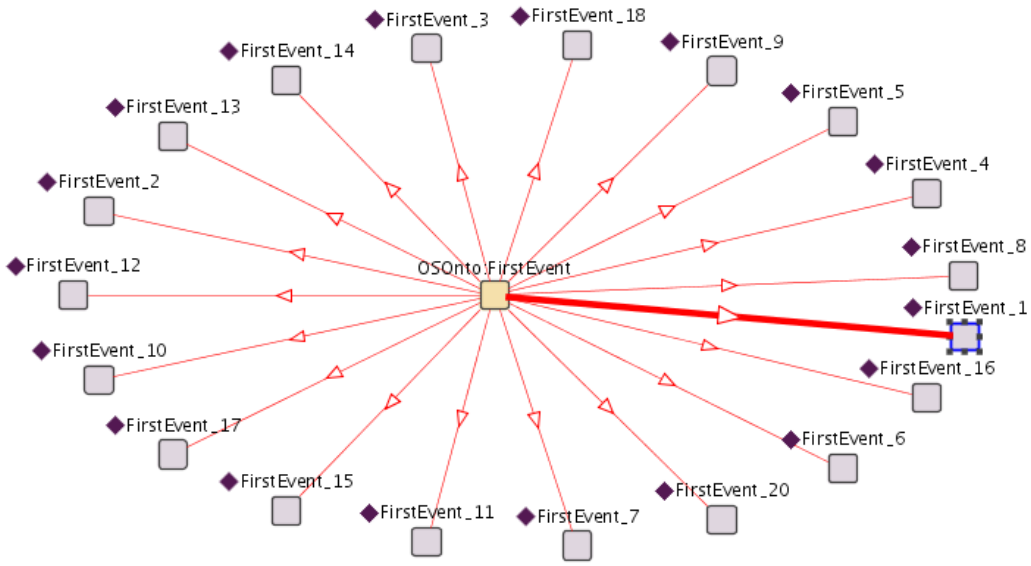

Figura A.13: A classe *FirstEvent* e suas suas instâncias

Condições Necessárias:

(i) **LastEvent** é subclasse de **Event**.

$$\texttt{LastEvent} \sqsubseteq \texttt{Event}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **LastEvent**.

$$\{\ \texttt{LastEvent\_1}\ \} \sqsubseteq \texttt{LastEvent}$$

A Figura A.14 mostra a relação da classe **LastEvent** com suas instâncias. *LastEvent_1* tem a seguinte descrição:

$$\texttt{hasEventDescription}\left(\texttt{LastEvent\_1}, \left(\begin{array}{c}\text{Sistema cria um}\\ \text{novo arquivo com}\\ \text{o nome que o usuário}\\ \text{passou como argumento}\end{array}\right)\right)$$

Este evento pertence a seguinte seqüência:

$$\texttt{isEventOf}(\texttt{LastEvent\_1}, \texttt{Tipical\_1})$$

LastEvent_1 tem como antecessor:

$$\texttt{hasPrevious}(\texttt{LastEvent\_1}, \texttt{SequenceEvent\_1})$$

- ***SequenceEvent:*** são aqueles eventos que tem um antecessor e um secessor.

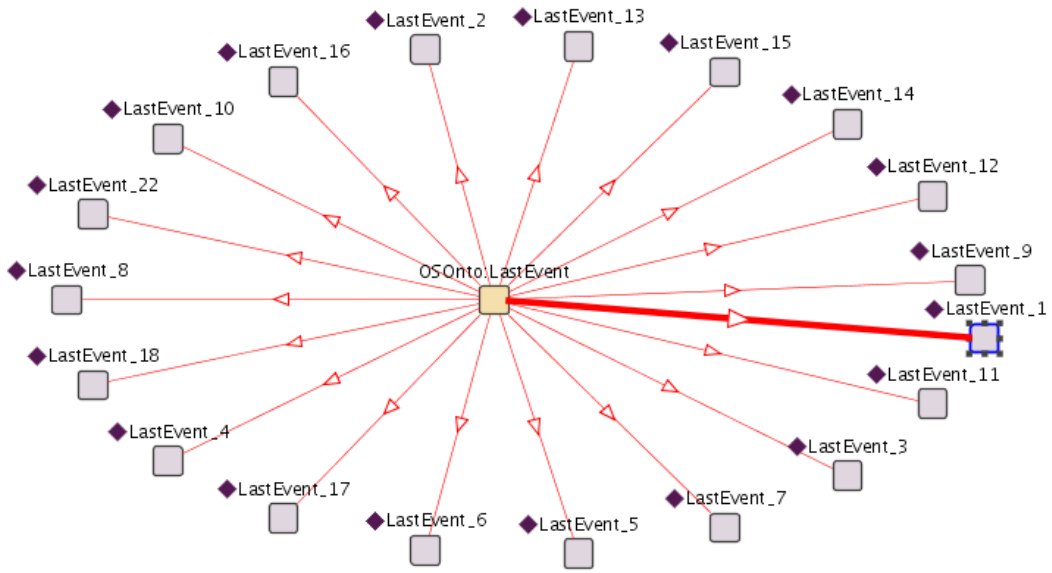

Figura A.14: A classe *LastEvent* e suas suas instâncias

Condição Necessária e Suficiente:

(i) Um indivíduo pode ser inferido como *SequenceEvent* se e somente se ele tiver exatamente um antecessor e extamente um sucessor. O evento antecessor a um evento seqüenciável pode ser o primeiro evento ou um outro evento seqüenciável. Um evento sucessor a um evento seqüenciável pode ser o último evento ou um outro evento sequenciável.

$$\texttt{SequenceEvent} \equiv \texttt{hasPrevious} = \texttt{1 Event}$$

$$\texttt{SequenceEvent} \equiv \texttt{hasSubsequent} = \texttt{1 Event}$$

Condições Necessárias:

(i) **SequenceEvent** é subclasse de **Event**.

$$\texttt{SequenceEvent} \sqsubseteq \texttt{Event}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **SequenceEvent**.

$$\{\ \texttt{SequenceEvent\_1}\ \} \sqsubseteq \texttt{SequenceEvent}$$

A Figura A.15 mostra a relação da classe **SequenceEvent** com suas instâncias. Este evento *SequenceEvent_1* tem a seguinte descrição:

$$\texttt{hasEventDescription}\left(\texttt{SequenceEvent\_1}, \left(\begin{array}{c}\text{Sistema verifica se usuário tem}\\ \text{permissão de escrita no diretório}\end{array}\right)\right)$$

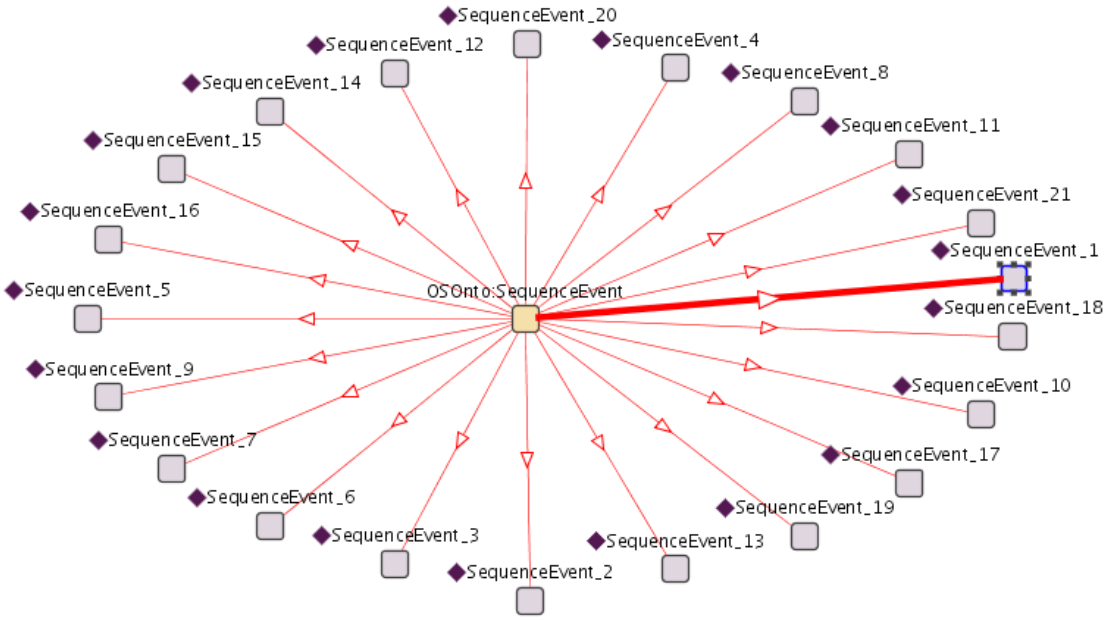

Figura A.15: A classe *SequenceEvent* e suas suas instâncias

Este evento pertence a seguinte seqüência:

$$\texttt{isEventOf}(\texttt{SequenceEvent\_1}, \texttt{Tipical\_1})$$

Este evento tem como antecessor:

$$\texttt{hasPrevious}(\texttt{SequenceEvent\_1}, \texttt{FirstEvent\_1})$$

E como sucessor:

$$\texttt{hasSubsequent}(\texttt{SequenceEvent\_1}, \texttt{LastEvent\_1})$$

- ***UnitEvent:*** são eventos unitários que não se relacionam com nenhum outro evento.
  Condição Necessária e Suficiente:

  (i) Um indivíduo pode ser inferido como *UnitEvent* se e somente se ele não tiver relação através da propriedade *hasPrevious* e *hasSubsequent* com indivíduos da classe *Event*.

$$\texttt{UnitEvent} \equiv \neg\,(\texttt{hasPrevious}\,.\,\texttt{Event})$$

$$\texttt{UnitEvent} \equiv \neg\,(\texttt{hasSubsequent}\,.\,\texttt{Event})$$

  Condições Necessárias:

  (i) **UnitEvent** é subclasse de **Event**.

$$\texttt{UnitEvent} \sqsubseteq \texttt{Event}$$

- ***EventSequence:*** esta classe representa os eventos ordenados que são registrados em uma seqüência.
  Condição Necessária e Suficiente:

  (i) Indivíduos da classe *EventSequence* são inferidos como tal se e somente se fizerem parte de uma das subclasses *Alternative* ou *Tipical*, já que estas são disjuntas.

$$\texttt{EventSequence} \equiv \texttt{Alternative} \sqcup \texttt{Tipical}$$

  Condições Necessárias:

  (i) **EventSequence** é subclasse de **DesignDocumentation**.

$$\texttt{EventSequence} \sqsubseteq \texttt{DesignDocumentation}$$

  (ii) Indivíduos da classe *EventSequence* só se relacionam através da propriedade *hasEvent* com indivíduos da classe *Event*.

$$\texttt{EventSequence} \sqsubseteq \forall\, \texttt{hasEvent.Event}$$

  (iii) Indivíduos da classe *EventSequence* se relacionam através da propriedade *hasEvent* com pelo menos um indivíduo da classe *Event*.

$$\texttt{EventSequence} \sqsubseteq \exists\, \texttt{hasEvent.Event}$$

  (iv) Indivíduos da classe *EventSequence* só se relacionam através da propriedade *isEventSequenceOf* com indivíduos da classe *Expanded*.

$$\texttt{EventSequence} \sqsubseteq \forall\, \texttt{isEventSequenceOf.Expanded}$$

  Propriedades - *Sobre as propriedades que descrevem a classe* ***EventSequence****, pode-se dizer:*

  (i) **hasEvent** é uma propriedade objeto inversa a **isEventOf**, cujo domínio é **EventSequence** e o contradomínio é **Event**.

$$\texttt{hasEvent} \in \texttt{P}_0 \tag{A.159}$$
$$\texttt{hasEvent} \equiv \texttt{isEventOf}^- \tag{A.160}$$
$$\top \sqsubseteq \forall\, \texttt{hasEvent} \,.\, \texttt{EventSequence} \tag{A.161}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasEvent}^- \,. \\ \texttt{Event} \end{pmatrix} \tag{A.162}$$

  (ii) **isEventSequenceOf** é uma propriedade objeto inversa a **hasEventSequence**, cujo domínio é **EventSequence** e o contradomínio é **Expanded**.

$$\texttt{isEventSequenceOf} \in \texttt{P}_0 \tag{A.163}$$
$$\texttt{isEventSequenceOf} \equiv \texttt{hasEventSequence}^- \tag{A.164}$$
$$\top \sqsubseteq \forall\, \texttt{isEventSequenceOf} \,.\, \texttt{EventSequence} \tag{A.165}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isEventSequenceOf}^- \,. \\ \texttt{Expanded} \end{pmatrix} \tag{A.166}$$

  Disjunções: - *Sobre as classes disjuntas de* ***EventSequence****, pode-se dizer:*

(i) A classe **EventSequence** é disjunta das classes **ImportanceRanking**, **ExternalBehavior**, **Actor**, **SoftwareRequirement** e **UseCase** já que estas classes não compartilham instâncias.

$$\texttt{EventSequence} \sqsubseteq \begin{pmatrix} \neg\ \texttt{ImportanceRanking}, \\ \neg\ \texttt{ExternalBehavior}, \\ \neg\ \texttt{Actor}, \\ \neg\ \texttt{SoftwareRequirement}, \\ \neg\ \texttt{UseCase} \end{pmatrix}$$

- ***Alternative:*** é uma seqüência de eventos que representa uma exceção. Esta seqüência detalha o que não faz parte do fluxo normal de execução ou operação.
Condição Necessária e Suficiente:

(i) Indivíduos serão inferidos como instâncias da classe *Alternative* se e somente se tiverem relação com outros indivíduos da classe *Tipical* através da propriedade *isExeptionOf* e com pelo menos uma seqüência típica.

$$\texttt{Alternative} \equiv \forall\ \texttt{isExceptionOf.TipicalAlternative} \equiv \exists\ \texttt{isExceptionOf.Tipical}$$

Condições Necessárias:

(i) **Alternative** é subclasse de **EventSequence**.

$$\texttt{Alternative} \sqsubseteq \texttt{EventSequence}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***Alternative****, pode-se dizer:*

(i) **isExceptionOf** é uma propriedade objeto inversa a **hasAlternative**, cujo domínio é **Alternative** e o contradomínio é **Tipical**.

$$\texttt{isExceptionOf} \in \texttt{P}_0 \tag{A.167}$$
$$\texttt{isExceptionOf} \equiv \texttt{hasAlternative}^- \tag{A.168}$$
$$\top \sqsubseteq \forall\ \texttt{isExceptionOf}\ .\ \texttt{Alternative} \tag{A.169}$$
$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isExceptionOf}^- . \\ \texttt{Tipical} \end{pmatrix} \tag{A.170}$$

Disjunções: - *Sobre as classes disjuntas de* ***Alternative****, pode-se dizer:*

(i) A classe **Alternative** é disjunta das classes **Tipical** já que estas classes não compartilham instâncias.

$$\texttt{Alternative} \sqsubseteq \neg\ \texttt{Tipical}$$

- ***Tipical:*** esta classe representa o conjunto das seqüências típicas dos casos de uso expandidos.
Condições Necessárias:

(i) **Tipical** é subclasse de **EventSequence**.

$$\texttt{Tipical} \sqsubseteq \texttt{EventSequence}$$

(ii) Indivíduos da classe *Tipical* só relacionam-se através da propriedade *hasAlternative* com indivíduos da classe *Alternative*.

$$\texttt{Tipical} \sqsubseteq \forall\, \texttt{hasAlternative.Alternative}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***Tipical****, pode-se dizer:*

(i) **hasAlternative** é uma propriedade objeto inversa a **isExceptionOf**, cujo domínio é **Tipical** e o contradomínio é **Alternative**.

$$\texttt{hasAlternative} \in \texttt{P}_0 \tag{A.171}$$

$$\texttt{hasAlternative} \equiv \texttt{isExceptionOf}^- \tag{A.172}$$

$$\top \sqsubseteq \forall\, \texttt{hasAlternative} \,.\, \texttt{Tipical} \tag{A.173}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasAlternative}^- \,. \\ \texttt{Alternative} \end{pmatrix} \tag{A.174}$$

Disjunções: - *Sobre as classes disjuntas de* ***Tipical****, pode-se dizer:*

(i) A classe **Tipical** é disjunta a classe **Alternative** já que estas classes não compartilham instâncias.

$$\texttt{Tipical} \sqsubseteq \neg\, \texttt{Alternative}$$

Instanciação: O exemplo a seguir demonstra instâncias da classe **Tipical**.

$$\{\, \texttt{Tipical\_1} \,\} \sqsubseteq \texttt{Tipical}$$

A Figura A.16 mostra a relação da classe **Tipical** com suas instâncias. Esta sequencia tipica tem os sequintes eventos:

$$\texttt{hasEvent}(\texttt{Tipical\_1}, \texttt{LastEvent\_1})$$
$$\texttt{hasEvent}(\texttt{Tipical\_1}, \texttt{SequenceEvent\_1})$$
$$\texttt{hasEvent}(\texttt{Tipical\_1}, \texttt{FirstEvent\_1})$$

Este sequencia de eventos faz parte do seguinte caso de uso expandido:

$$\texttt{isEventSequenceOf}(\texttt{Tipical\_1}, \texttt{CreateEmptyFile})$$

- ***Frontier:*** é importante definir as fronteiras do sistema para identificar quais são os comportamentos externos e internos bem como as responsabilidades do sistema. A escolha da fronteira do sistema é influenciada pelas necessidades e especificação de requisitos.
Condição Necessária e Suficiente:

(i) Toda fonteira delimita um comportamento externo ou interno. Desta forma, instâncias da classe *Frontier* também participam de uma de suas subclasses *ExternalBehavior* ou *InternalBehavior* já que elas são disjuntas.

$$\texttt{Frontier} \equiv \texttt{ExternalBehavior} \sqcup \texttt{InternalBehavior}$$

Condições Necessárias:

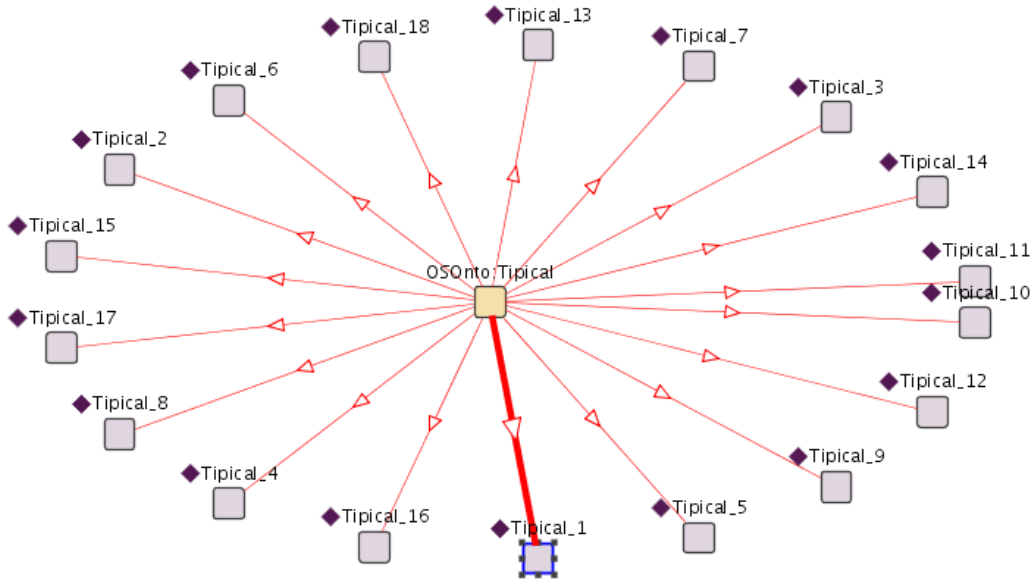

Figura A.16: A classe *Tipical* e suas suas instâncias

(i) **Frontier** é subclasse de **DesignDocumentation**.

$$\texttt{Frontier} \sqsubseteq \texttt{DesignDocumentation}$$

Disjunções: - *Sobre as classes disjuntas de* ***Frontier****, pode-se dizer:*

(i) A classe **Frontier** é disjunta das classes **SoftwareRequirement**, **ImportanceRanking**, **UseCase** e **Event** já que estas classes não compartilham instâncias.

$$\texttt{Alternative} \sqsubseteq \begin{pmatrix} \neg\, \texttt{SoftwareRequirement} \sqcap \neg\, \texttt{ImportanceRanking} \sqcap \\ \neg\, \texttt{UseCase} \sqcap \neg\, \texttt{Event} \end{pmatrix}$$

- ***ExternalBehavior:*** esta classe representa comportamentos externos ao sistema.
Condições Necessárias:

(i) **ExternalBehavior** é subclasse de **Frontier**.

$$\texttt{ExternalBehavior} \sqsubseteq \texttt{Frontier}$$

(ii) Indivíduos da classe *ExternalBehavior* só relacionam-se através da propriedade *delimits* com indivíduos da classe *Actor*.

$$\texttt{ExternalBehavior} \sqsubseteq \forall\, \texttt{delimits}\, .\, \texttt{Actor}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***ExternalBehavior****, pode-se dizer:*

(i) **delimits** é uma propriedade objeto inversa a **isDelimitedBy**.

$$\texttt{delimits} \in \texttt{P}_0 \tag{A.175}$$

$$\texttt{delimits} \equiv \texttt{isDelimitedBy}^- \tag{A.176}$$

Disjunções: - *Sobre as classes disjuntas de* ***ExternalBehavior****, pode-se dizer:*

(i) A classe **ExternalBehavior** é disjunta das classes **InternalBehavior**, **UseCase**, **EventSequence** e **Event** já que estas classes não compartilham instâncias.

$$\texttt{ExternalBehavior} \sqsubseteq \begin{pmatrix} \neg\ \texttt{InternalBehavior} \sqcap \neg\ \texttt{UseCase} \sqcap \\ \neg\ \texttt{EventSequence} \sqcap \neg\ \texttt{Event} \end{pmatrix}$$

- ***InternalBehavior:*** esta classe representa o comportamento interno do sistema.
Condições Necessárias:

(i) **InternalBehavior** é subclasse de **Frontier**.

$$\texttt{InternalBehavior} \sqsubseteq \texttt{Frontier}$$

(ii) Indivíduos da classe *InternalBehavior* só relacionam-se através da propriedade *delimits* com indivíduos da classe `UseCase`.

$$\texttt{InternalBehavior} \sqsubseteq \forall\ \texttt{delimits}\ .\ \texttt{UseCase}$$

Disjunções: - *Sobre as classes disjuntas de* ***InternalBehavior****, pode-se dizer:*

(i) A classe **InternalBehavior** é disjunta das classes **ExternalBehavior** e **Actor** já que estas classes não compartilham instâncias.

$$\texttt{InternalBehavior} \sqsubseteq \neg\ \texttt{ExternalBehavior} \sqcap \neg\ \texttt{Actor}$$

- ***ImportanceRanking:*** os casos de uso podem ser classificados de acordo com o seu grau de importância para o projeto. Este grau de importância define se o caso de uso é primário ou secundário.
Condição Necessária e Suficiente:

(i) A classe **ImportanceRanking** é uma classe definida e numerada, portanto contém somente os indivíduos primário e secundário.

$$\texttt{ImportanceRanking} \equiv \{\ \texttt{Primary}\ ,\ \texttt{Secondary}\ \}$$

Condições Necessárias:

(i) **ImportanceRanking** é subclasse de **DesignDocumentation**.

$$\texttt{ImportanceRanking} \sqsubseteq \texttt{DesignDocumentation}$$

Disjunções: - *Sobre as classes disjuntas de* ***ImportanceRanking****, pode-se dizer:*

(i) A classe **ImportanceRanking** é disjunta das classes **Frontier**, **SoftwareRequirement**, **UseCase**, **Actor**, **Event**, **Scenario**, **EventSequence** e **IntendedBehavior**

já que estas classes não compartilham instâncias.

$$\texttt{ImportanceRanking} \sqsubseteq \begin{pmatrix} \neg\ \texttt{Frontier}, \\ \neg\ \texttt{SoftwareRequirement}, \\ \neg\ \texttt{UseCase}, \\ \neg\ \texttt{Actor}, \\ \neg\ \texttt{Event}, \\ \neg\ \texttt{Scenario}, \\ \neg\ \texttt{EventSequence}, \\ \neg\ \texttt{IntendedBehavior}, \end{pmatrix}$$

- ***IntendedBehavior:*** trata-se de uma compreensão parcial dos requisitos do sistema. Esta compreensão pode ser expressa por serviços, tarefas e funções que o sistema deve desempenhar.
  Condições Necessárias:

  (i) **IntendedBehavior** é subclasse de **DesignDocumentation**.

  $$\texttt{IntendedBehavior} \sqsubseteq \texttt{DesignDocumentation}$$

  (ii) Indivíduos da classe *IntendedBehavior* só relacionam-se através da propriedade *isPartOf* com indivíduos da classe *Scenario*.

  $$\texttt{IntendedBehavior} \sqsubseteq \forall\ \texttt{isPartOf.Scenario}$$

  Disjunções: - *Sobre as classes disjuntas de* ***IntendedBehavior****, pode-se dizer:*

  (i) A classe **IntendedBehavior** é disjunta a classes **ImportanceRanking** já que estas classes não compartilham instâncias.

  $$\texttt{IntendedBehavior} \sqsubseteq \neg\ \texttt{ImportanceRanking}$$

  A classe **IntendedBehavior** possui três subclasses que a princípio não são disjuntas entre si:

  - ***Function:*** representa o emprego, uso ou papel que o sistema deve desempenhar.
  - ***Service:*** representa o conjunto de serviços que o sistema deve realizar.
  - ***Task:*** representa o conjunto de tarefas que devem ser realizada e concluídas em tempo hábil pelo sistema.

- ***Scenario:*** é uma forma prática para contextualizar a elicitação de requisitos. Os senários são utilizados pelos engenheiros de *software* como um *framework*, onde questionários (sobre atividades dos usuários, por exemplo) são elaborados com o intuito de descobrir "o que é"e "como as coisas são feitas".
  Condições Necessárias:

  (i) **Scenario** é subclasse de **DesignDocumentation**.

  $$\texttt{Scenario} \sqsubseteq \texttt{DesignDocumentation}$$

  (ii) Um cenário representa parte da compreensão parcial do sistema. Desta forma, indivíduos da classe *Scenario* só relacionam-se através da propriedade *hasPart* com indivíduos da classe *IntendedBehavior*

  $$\texttt{Scenario} \sqsubseteq \forall\ \texttt{hasPart.IntendedBehavior}$$

(iii) Um cenário é parte de um caso de uso. Desta forma, indivíduos da classe *Scenario* só relacionam-se através da propriedade *isPartOf* com indivíduos da classe *UseCase*

$$\texttt{Scenario} \sqsubseteq \forall\ \texttt{isPartOf.UseCase}$$

Disjunções: - *Sobre as classes disjuntas de* ***Scenario****, pode-se dizer:*

(i) A classe **Scenario** é disjunta a classes **ImportanceRanking** já que estas classes não compartilham instâncias.

$$\texttt{Scenario} \sqsubseteq \neg\ \texttt{ImportanceRanking}$$

- ***SoftwareRequirement:*** é uma documentação, onde freqüentemente os requisitos são pesquisados e organizados por ordem de solução do problem. Uma distinção entre os requisitos pode ser definida entre os funcionais e os não funcionais.
Condição Necessária e Suficiente:

(i) Indivíduos podem ser inferidos como instâncias da classe *SoftwareRequirement* se e somente se participarem de uma das subclasses *FuncionalRequirement* ou *NonFuncionalRequirement* já que estas classes são disjuntas entre si.

$$\texttt{SoftwareRequirement} \equiv \texttt{FuncionalRequirement} \sqcup \texttt{NonFuncionalRequirement}$$

Condições Necessárias:

(i) **SoftwareRequirement** é subclasse de **DesignDocumentation**.

$$\texttt{SoftwareRequirement} \sqsubseteq \texttt{DesignDocumentation}$$

Disjunções: - *Sobre as classes disjuntas de* ***SoftwareRequirement****, pode-se dizer:*

(i) A classe **SoftwareRequirement** é disjunta das classes **UseCase**, **ImportanceRanking**, **Frontier**, **EventSequence**, **Event** e **Actor** já que estas classes não compartilham instâncias.

$$\texttt{SoftwareRequirement} \sqsubseteq \begin{pmatrix} \neg\ \texttt{UseCase}, \\ \neg\ \texttt{ImportanceRanking}, \\ \neg\ \texttt{Frontier}, \\ \neg\ \texttt{EventSequence}, \\ \neg\ \texttt{Event}, \\ \neg\ \texttt{Actor}, \end{pmatrix}$$

A figura A.17 introduz as subclasses de *SoftwareRequirement*. As classe *FunctionalRequirement* e *NonFunctionalRequirement* são disjuntas.

- ***UseCase:*** é um tipo de documentação onde cada caso de uso se apoia em um ou mais cenários para definir como o sistema deve interagir com seus atores para atender um objetivo como por exemplo, a realização de uma tarefa.
Condição Necessária e Suficiente:

(i) Indivíduos são inferidos como instâncias de *UseCase* se e somente se forem instâncias de uma das subclasses *Expanded* ou *HighLevel* já que as subclasses são disjuntas.

$$\texttt{UseCase} \equiv \texttt{Expanded} \sqcup \texttt{HighLevel}$$

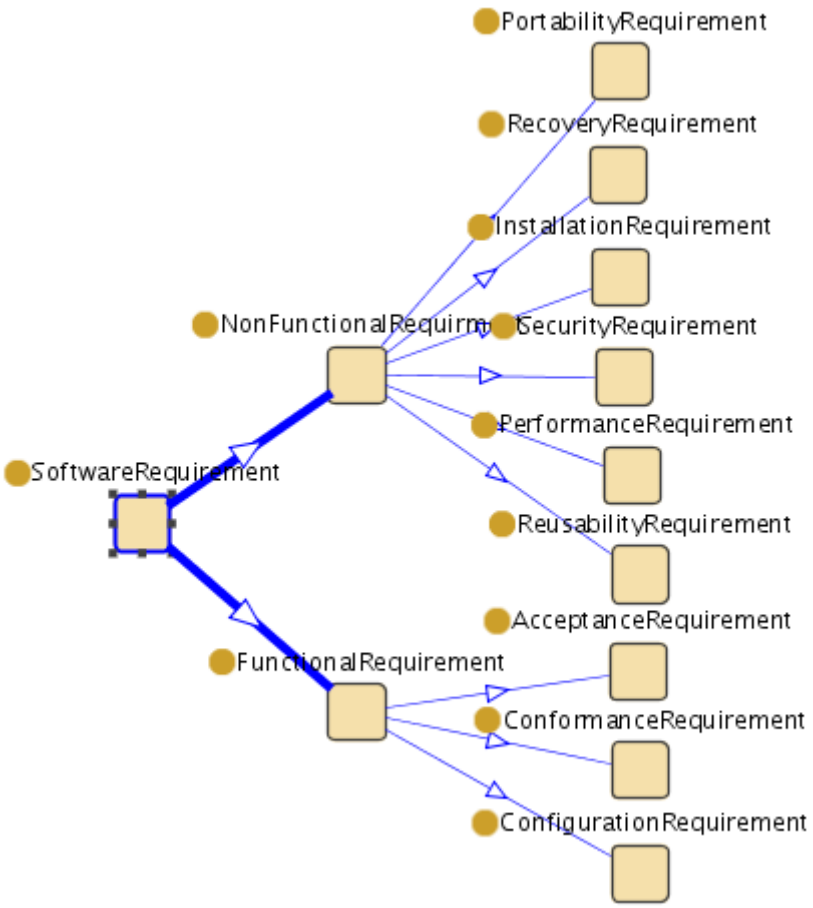

Figura A.17: A classe *SoftwareRequirement* e suas subclasses

Condições Necessárias:

(i) **UseCase** é subclasse de **DesignDocumentation**.

$$\texttt{UseCase} \sqsubseteq \texttt{DesignDocumentation}$$

(ii) Indivíduos da classe *UseCase* só relacionam-se através da propriedade *hasActor* com indivíduos da classe *Actor*.

$$\texttt{UseCase} \sqsubseteq \forall\, \texttt{hasActor.Actor}$$

(iii) Indivíduos da classe *UseCase* relacionam-se através da propriedade *hasActor* com pelo menos um indivíduo da classe *Actor*.

$$\texttt{UseCase} \sqsubseteq \exists\, \texttt{hasActor.Actor}$$

(iv) Indivíduos da classe *UseCase* só relacionam-se através da propriedade *hasRanking* com indivíduos da classe *ImportanceRanking*.

$$\texttt{UseCase} \sqsubseteq \forall\, \texttt{hasRanking.ImportanceRanking}$$

(v) Indivíduos da classe *UseCase* relacionam-se através da propriedade *hasRanking* com pelo menos um indivíduo da classe *ImportanceRanking*.

$$\texttt{UseCase} \sqsubseteq \exists\, \texttt{hasRanking.ImportanceRanking}$$

(vi) Indivíduos da classe *UseCase* só relacionam-se através da propriedade *isRelatedTo* com indivíduos da classe *UseCase*.

$$\texttt{UseCase} \sqsubseteq \forall\, \texttt{isRelatedTo.UseCase}$$

(vii) Indivíduos da classe *UseCase* só relacionam-se através da propriedade *isStartedBy* com indivíduos da classe *Initiator*.

$$\texttt{UseCase} \sqsubseteq \forall\, \texttt{isStartedBy.Initiator}$$

Propriedades - *Sobre as propriedades que descrevem a classe* ***UseCase****, pode-se dizer:*

(i) **hasActor** é uma propriedade objeto inversa a **isActorOf**, cujo domínio é **UseCase** e o contradomínio é **Actor**.

$$\begin{aligned} \texttt{hasActor} &\in \texttt{P}_0 \\ \texttt{hasActor} \equiv \texttt{isActorOf}^{-} \top &\sqsubseteq \forall\, \texttt{hasActor} \,.\, \texttt{UseCase} \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasActor}^{-} \,. \\ \texttt{Actor} \end{pmatrix} \end{aligned}$$

(ii) **hasRanking** é uma propriedade objeto funcional, cujo domínio é **UseCase** e o contradomínio é **ImportanceRanking**.

$$\begin{aligned} \texttt{hasRanking} &\in \texttt{P}_0 \\ \top &\sqsubseteq (\leq\, 1\ \texttt{hasRanking}) \\ \top &\sqsubseteq \forall\, \texttt{hasRanking} \,.\, \texttt{UseCase} \\ \top &\sqsubseteq \forall \begin{pmatrix} \texttt{hasRanking}^{-} \,. \\ \texttt{ImportanceRanking} \end{pmatrix} \end{aligned}$$

(iii) **isRelatedTo** é uma propriedade objeto, cujo domínio e contradomínio é **UseCase**.

$$\texttt{isRelatedTo} \in \texttt{P}_0 \tag{A.177}$$

$$\top \sqsubseteq \forall\, \texttt{isRelatedTo} \,.\, \texttt{UseCase} \tag{A.178}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isRelatedTo}^{-} \,. \\ \texttt{UseCase} \end{pmatrix} \tag{A.179}$$

(iv) **isStartedBy** é uma propriedade objeto inversa a propriedade **startsAn**, cujo domínio é a classe **UseCase** e contradomínio é **Initiator**.

$$\texttt{isRelatedTo} \in \texttt{P}_0 \tag{A.180}$$

$$\texttt{isRelatedTo} \equiv \texttt{startsAn}^{-} \tag{A.181}$$

$$\top \sqsubseteq \forall\, \texttt{isRelatedTo} \,.\, \texttt{UseCase} \tag{A.182}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{isRelatedTo}^{-} \,. \\ \texttt{UseCase} \end{pmatrix} \tag{A.183}$$

(v) Todo caso de uso tem um objetivo bem definido. A propriedade **hasGoal** representa este objetivo. Trata-se de uma propriedade de tipo de dados (Equação A.184)

cujo domínio é a classe **UseCase** (Equação A.185) e o contradomínio é **XMLSchema:string** (Equação A.186).

$$\texttt{hasGoal} \in \texttt{P}_\texttt{D} \tag{A.184}$$

$$\top \sqsubseteq \forall\, \texttt{hasGoal} \,.\, \texttt{UseCase}\ (\texttt{UseCase} \neq \bot) \tag{A.185}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasGoal}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \tag{A.186}$$

(vi) Um caso de uso pode ter uma condição que antecede a sua realização. A propriedade **hasPreCondition** representa esta condição. Trata-se de uma propriedade de tipo de dados (Equação A.187) cujo domínio é a classe **UseCase** (Equação A.188) e o contradomínio é **XMLSchema:string** (Equação A.189).

$$\texttt{hasPreCondition} \in \texttt{P}_\texttt{D} \tag{A.187}$$

$$\top \sqsubseteq \forall\, \texttt{hasPreCondition} \,.\, \texttt{UseCase}\ (\texttt{UseCase} \neq \bot) \tag{A.188}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasPreCondition}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \tag{A.189}$$

(vii) Um caso de uso pode ter uma condição que sucede a sua realização. A propriedade **hasPostCondition** representa esta condição. Trata-se de uma propriedade de tipo de dados (Equação A.190) cujo domínio é a classe **UseCase** (Equação A.191) e o contradomínio é **XMLSchema:string** (Equação A.192).

$$\texttt{hasPostCondition} \in \texttt{P}_\texttt{D} \tag{A.190}$$

$$\top \sqsubseteq \forall\, \texttt{hasPostCondition} \,.\, \texttt{UseCase}\ (\texttt{UseCase} \neq \bot) \tag{A.191}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasPostCondition}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \tag{A.192}$$

(viii) Para facilitar a organização dos casos de uso, eles podem dispor de um número de identificação. A propriedade **hasUseCaseIdNumber** representa esta identificação. Trata-se de uma propriedade de tipo de dados (Equação A.193) funcional, cujo domínio é a classe **UseCase** (Equação A.195) e o contradomínio é **XMLSchema:int** (Equação A.196).

$$\texttt{hasUseCaseIdNumber} \in \texttt{P}_\texttt{D} \tag{A.193}$$

$$\top \sqsubseteq (\leq\ 1\ \texttt{hasUseCaseIdNumber}) \tag{A.194}$$

$$\top \sqsubseteq \forall\, \texttt{hasUseCaseIdNumber} \,.\, \texttt{UseCase}\ (\texttt{UseCase} \neq \bot) \tag{A.195}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasUseCaseIdNumber}^- \,.\, \texttt{XMLSchema} : \texttt{int} \\ (\texttt{XMLSchema} : \texttt{int} \neq \bot) \end{pmatrix} \tag{A.196}$$

(ix) Para facilitar a identificação dos casos de uso, eles podem dispor de um nome. A propriedade **hasUseCaseName** representa esta identificação. Trata-se de uma propriedade de tipo de dados (Equação A.197) funcional, cujo domínio é a classe **UseCase** (Equação A.199) e o contradomínio é **XMLSchema:string** (Equação A.200).

$$\texttt{hasUseCaseName} \in \texttt{P}_\texttt{D} \tag{A.197}$$

$$\top \sqsubseteq (\leq\ 1\ \texttt{hasUseCaseName}) \tag{A.198}$$

$$\top \sqsubseteq \forall\, \texttt{hasUseCaseName} \,.\, \texttt{UseCase}\ (\texttt{UseCase} \neq \bot) \tag{A.199}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasUseCaseName}^- \,.\, \texttt{XMLSchema} : \texttt{string} \\ (\texttt{XMLSchema} : \texttt{string} \neq \bot) \end{pmatrix} \tag{A.200}$$

(x) Por questões de projeto, os casos de uso podem ter prioridade. A propriedade **hasUseCasePriority** representa esta prioridade. Trata-se de uma propriedade de tipo de dados (Equação A.201) funcional, cujo domínio é a classe **UseCase** (Equação A.203) e o contradomínio é **XMLSchema:int** (Equação A.204).

$$\texttt{hasUseCasePriority} \in \texttt{P}_\texttt{D} \tag{A.201}$$

$$\top \sqsubseteq (\leq 1\ \texttt{hasUseCasePriority}) \tag{A.202}$$

$$\top \sqsubseteq \forall\, \texttt{hasUseCasePriority} \,.\, \texttt{UseCase}\ (\texttt{UseCase} \neq \bot) \tag{A.203}$$

$$\top \sqsubseteq \forall \begin{pmatrix} \texttt{hasUseCasePriority}^- \,.\, \texttt{XMLSchema:int} \\ (\texttt{XMLSchema:int} \neq \bot) \end{pmatrix} \tag{A.204}$$

Essa propriedade tem alguns valores inteiros definidos pelo construtor owl:oneOf onde um caso de uso pode ter prioridade de 1 a 5, onde 1 significa baixa prioridade e 5 alta prioridade.

$$\{\ 1, 2, 3, 4, 5\ \} \sqsubseteq \texttt{hasUseCasePriority}$$

Disjunções: - *Sobre as classes disjuntas de* ***useCase****, pode-se dizer:*

(i) A classe **UseCase** é disjunta das classes **EventSequence**, **Actor**, **ImportanceRanking**, **ExternalBehavior**, **SoftwareRequirement**, **Event** e **Frontier** já que estas classes não compartilham instâncias.

$$\texttt{Event} \sqsubseteq \begin{pmatrix} \neg\, \texttt{EventSequence}, \\ \neg\, \texttt{Actor}, \\ \neg\, \texttt{ImportanceRanking}, \\ \neg\, \texttt{ExternalBehavior}, \\ \neg\, \texttt{SoftwareRequirement}, \\ \neg\, \texttt{Event}, \\ \neg\, \texttt{Frontier}, \end{pmatrix}$$

A classe **UseCase** possui *Expanded* e *HighLevel* como subclasses que são disjuntas. A classe *HighLevel* representa os casos de uso que não tem eventos. A classe *Expanded*, detalhada a seguir, representa os casos de uso que tem eventos.

- ***Expanded:*** esta classe representa o conjunto dos casos de uso expandidos. Um caso de uso expandido é aquele que tem seqüência de eventos.
Condição Necessária e Suficiente:

(i) Indivíduos são inferidos como instâncias da classe *Expanded* se e somente se relacionarem através da propriedade *hasEventSequence* com indivíduos da classe *EventSequence*.

$$\texttt{Expanded} \equiv \forall\, \texttt{hasEventSequence} \,.\, \texttt{EventSequence})$$

$$\texttt{Expanded} \equiv \exists\, \texttt{hasEventSequence} \,.\, \texttt{EventSequence})$$

Instanciação: A Figura A.18 mostra a relação da classe **Expanded** com suas instâncias.

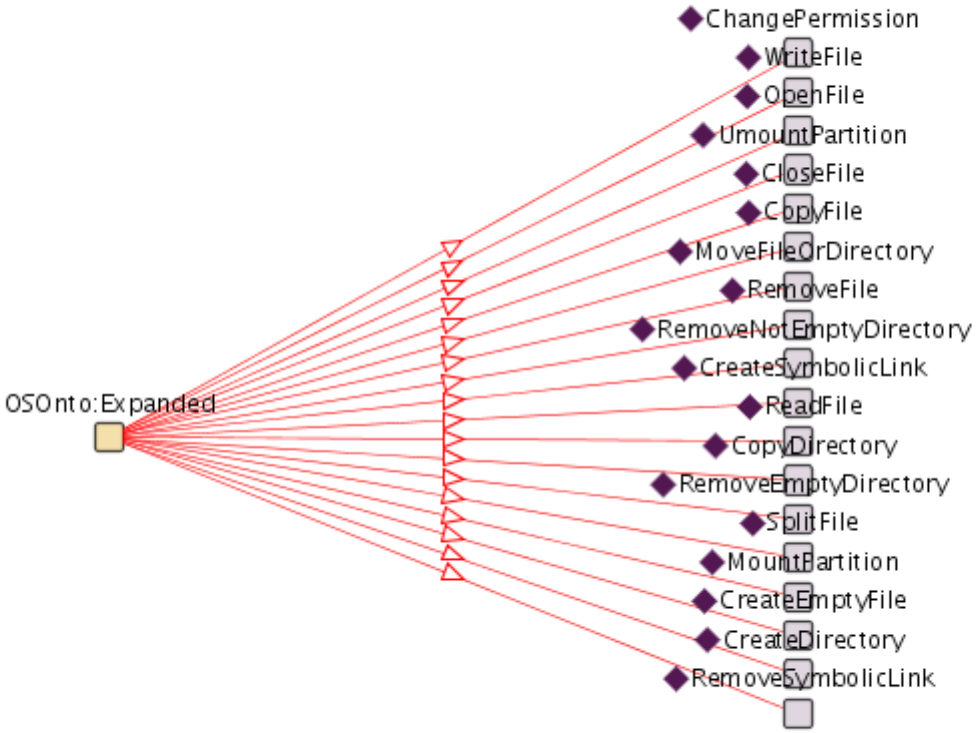

Figura A.18: A classe *Expanded* e suas suas instâncias