REESTRUTURANDO ESPECIFICAÇÕES DE RESTRIÇÕES DE MODELOS ELABORADAS EM OCL

Alexandre Luis Correa

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Profa. Cláudia Maria Lima Werner, D.Sc.

Prof. Guilherme Horta Travassos, D.Sc.

Prof. Gerson Zaverucha, Ph.D.

Prof. Augusto Cesar Alves Sampaio, D.Phil.

Prof. Julio Cesar Sampaio do Prado Leite, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

MAIO DE 2006

CORREA, ALEXANDRE LUIS

Reestruturando Especificações de Restrições de Modelos Elaboradas em OCL [Rio de Janeiro] 2006

XIX, 252p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2006)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Especificações de Software

2. Reestruturação de Especificações

3. Restrições em OCL

I. COPPE/UFRJ II. Título (série)

Dedico esta tese à minha esposa *Glória*,

à minha filha *Caroline*

e aos meus pais *Paulo* e *Dirce*,

pelo amor, apoio e compreensão em todos os momentos.

Agradecimentos

Esta tese foi resultado de uma jornada de seis anos de trabalho, e não teria sido bem sucedida sem o apoio, a participação e a colaboração de diversas pessoas, às quais eu gostaria de prestar meus sinceros agradecimentos.

Antes, porém, gostaria de agradecer a Deus por ter iluminado o meu caminho nos momentos em que tudo parecia entregue às trevas, e por ter me dado forças para superar obstáculos e desafios aparentemente intransponíveis.

À Profa. Cláudia Werner, não apenas pela orientação precisa, mas sobretudo pela dedicação, amizade, paciência, compreensão e confiança, aceitando um trabalho de pesquisa em uma área um pouco distante da sua linha principal de trabalho. Suas críticas, sempre construtivas, aliadas às palavras de incentivo e apoio nos momentos mais importantes, foram absolutamente decisivas para a geração desta tese.

Ao Prof. Guilherme Travassos, não somente pela contribuição ao meu aprendizado no decorrer dos cursos de Mestrado e Doutorado, mas também pelas valiosas críticas recebidas no exame de qualificação e pela participação na banca examinadora desta tese.

Ao Prof. Gerson Zaverucha, pela contribuição ao meu aprendizado no decorrer do curso de Mestrado, pelas palavras de incentivo e entusiasmo para que eu aceitasse este desafio, assim como pelo incentivo demonstrado em diversos momentos ao longo deste trabalho. Agradeço, também, pela participação na banca examinadora desta tese.

Ao Prof. Julio Leite, pelas valiosas críticas recebidas no exame de qualificação e pela participação na banca examinadora desta tese.

Ao Prof. Augusto Sampaio, pela participação na banca examinadora desta tese.

Aos meus pais, pelo amor, compreensão e, sobretudo, por terem plantado em mim a semente da busca permanente pelo conhecimento.

À minha esposa Glória, que sempre esteve ao meu lado nesta caminhada tão difícil, com muito amor, compreensão e sacrifício, sobretudo nos últimos meses deste trabalho.

À minha filha Caroline, nascida durante esta jornada, pelo seu amor incondicional, pela alegria que trouxe ao nosso lar e pelos sorrisos que iluminaram minha alma, mesmo tendo ficado, por muitas vezes, privada de uma dedicação paterna plena.

Ao meu sogro Francisco e à minha sogra Cidália, pelo apoio dado ao longo deste período, em especial, nos últimos meses.

Aos professores e, sobretudo, amigos Éber Schmitz e Ricardo Bianchini, pelo permanente incentivo para que eu me dedicasse à área de ensino e pesquisa.

Ao amigo Márcio Barros, pela demonstração inequívoca de amizade e solidariedade, e pelas valiosas contribuições ao estudo experimental realizado.

Um agradecimento especial a todos aqueles que participaram do estudo experimental, não apenas pelo precioso tempo dedicado, mas também pelo empenho e seriedade com que participaram de todas as etapas desse estudo.

Aos amigos Fernando Farias e João Carlos Ribeiro, pelo inestimável apoio, especialmente nos últimos meses deste trabalho.

A todos os amigos da COPPE, incluindo os que já se formaram e os que ainda estão por se formar, pelo excelente ambiente de trabalho e pela amizade. Em especial, agradeço aos amigos Alexandre Dantas, Aline Vasconcelos, Ana Paula Blois, Carlos Melo Junior, Cristine Dantas, Denis Silveira, Gustavo Veronese, Hamilton Oliveira, Hugo Teixeira, Isabella Silva, José Ricardo Xavier, Leonardo Murta, Luiz Gustavo Lopes, Marco Lopes, Marco Mangan, Natanael Maia, Regiane Oliveira e Regina Braga.

A outros pesquisadores dedicados ao estudo da OCL, como Thomas Baar, Martin Gogolla, Heinrich Hussmann, Jos Warmer, Martin Giese e Peter Schmitt, por suas valiosas críticas que contribuíram sobremaneira para o desenvolvimento deste trabalho, e também pela oportunidade de participar do comitê de programa do *Workshop on Tool Support for OCL and Related Formalisms*, realizado em 2005.

Ao comitê de programa da conferência UML'04, atualmente denominada MoDELS, pelo prêmio concedido ao nosso artigo, que nos deu uma motivação adicional para a realização deste trabalho.

Ao pessoal da secretaria do Departamento de Engenharia de Sistemas e Computação da COPPE/UFRJ, pelo apoio e pelos serviços prestados ao longo de todo o período.

Ao Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, pelo apoio financeiro nas conferências internacionais das quais eu participei.

Ao CNPq, pela bolsa concedida no início deste trabalho.

Finalmente, agradeço a todos que estiveram torcendo por mim ao longo deste período

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

REESTRUTURANDO ESPECIFICAÇÕES DE RESTRIÇÕES DE MODELOS ELABORADAS EM OCL

Alexandre Luis Correa

Maio/2006

Orientadora: Cláudia Maria Lima Werner

Programa: Engenharia de Sistemas e Computação

A OCL (Object Constraint Language) é uma linguagem que pode ser utilizada para especificar, de forma precisa, elementos que as notações gráficas da UML não são capazes de representar, como, por exemplo, restrições, expressões associadas a atributos derivados, expressões de consulta e definições contratuais de operações. Embora a OCL tenha sido definida com o objetivo de ser uma linguagem de uso mais fácil, se comparada às linguagens formais tradicionais, especificações produzidas com a OCL podem apresentar problemas de legibilidade e manutenibilidade.

A partir de um estudo realizado em diversas especificações, identificamos e catalogamos construções potencialmente problemáticas ao entendimento e à manutenção de especificações que empreguem a OCL. Um conjunto de reestruturações foi proposto com o objetivo de apoiar a substituição dessas construções por outras mais adequadas. Um estudo experimental foi realizado para avaliar a viabilidade de aplicação destes conceitos. Em particular, foi feita uma avaliação preliminar de como a compreensão de restrições especificadas em OCL pode ser afetada pela estrutura das expressões que a compõem. Finalmente, este trabalho propõe uma abordagem e um software para apoiar a automação de reestruturações e a verificação da preservação da semântica do modelo no caso de reestruturações realizadas manualmente.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

REFACTORING OCL MODEL CONSTRAINTS SPECIFICATIONS

Alexandre Luis Correa

May/2006

Advisor: Cláudia Maria Lima Werner

Department: Computer and Systems Engineering

OCL (Object Constraint Language) is a specification language that can be used to precisely specify elements that can not be represented by the graphical elements of UML. Constraints, derivation rules, query expressions, operation contracts are some examples of such elements. Although OCL has been designed to be a simpler language when compared to traditional formal specification languages, OCL specifications may be difficult to understand and evolve.

From a study performed on several OCL specifications, we have identified a set of constructions that are hints that some part of an OCL specification, or even of the underlying model, should be refactored in order to make it easier to understand and maintain. A set of refactorings has been proposed in order to replace such constructions by more adequate ones. An experimental study has been performed to evaluate the applicability of those concepts. This study presents the results of a preliminary evaluation of the effects of the structure of OCL expressions on its understandability. Moreover, we present an approach and tool support for refactoring automation and to manually performed refactorings.

Índice

Índice de Figuras

Índice de Tabelas

Capítulo 1

Introdução

1.1 Motivação

A elaboração de modelos é uma técnica amplamente empregada em diversas modalidades de engenharia (HAZELRIGG, 1999). Na Engenharia de Software, modelos podem ser elaborados em diversas atividades e em diferentes estágios do desenvolvimento de sistemas (PRESSMAN, 2004). Modelos de sistemas possibilitam a visualização, a comunicação e a validação de aspectos de um sistema antes da sua construção. Um modelo de sistema é representado em uma linguagem de modelagem, que pode empregar uma combinação de representações gráficas e textuais (RUMBAUGH et al., 2004).

No contexto de desenvolvimento de software orientado a objetos ou baseado em componentes, a UML - *Unified Modeling Language* - (BOOCH et al., 2005) é, atualmente, o padrão *de facto* para a elaboração de modelos, sendo bastante difundida tanto na indústria como na academia. Resultado do processo de unificação de linguagens de modelagem propostas por diversos autores (RUMBAUGH, 1990), (JACOBSON, 1992) e (BOOCH, 1994), a UML foi adotada como padrão pelo OMG - Object Management Group - (OMG, 1999) que, desde então, assumiu a responsabilidade por sua evolução.

A forma mais comum de utilização da UML consiste na elaboração de modelos contendo representações gráficas semi-formais, empregando os diversos diagramas definidos pela linguagem, complementadas com anotações elaboradas em linguagem natural. Em cenários que demandam maior precisão do modelo, a OCL - *Object Constraint Language* - (WARMER e KLEPPE, 2003) é uma linguagem que pode ser utilizada para especificar, de forma precisa, elementos que as notações gráficas da UML não são capazes de representar, como, por exemplo, restrições, expressões associadas a atributos derivados e expressões de consulta. Em sua primeira versão, a OCL era parte integrante da especificação da UML, mas, a partir da versão 2.0, ela passou a ter uma

especificação própria (OMG, 2003a).

A OCL vem sendo empregada na elaboração de modelos precisos por métodos de desenvolvimento baseado em componentes como Catalysis (D'SOUZA e WILLS, 1998) e UML Components (CHEESMAN e DANIELS, 2001), na especificação de requisitos funcionais de software (TORO, 2000), (SENDALL, 2002), (CORREA e WERNER, 2004b), na definição de metamodelos (OMG, 2002a), (FERNANDEZ-MEDINA e PIATTINI, 2004), na transformação de modelos (CARIOU et al., 2004), (OMG, 2004a), como linguagem de consulta para a extração de métricas de modelos (BARONI e ABREU, 2002), entre outras aplicações. Os benefícios que podem advir da elaboração de especificações precisas são relatados em diversos trabalhos como, por exemplo, (HALL, 1996), (SATPATHY et al., 2001) e (BRIAND et al., 2005).

A OCL é uma linguagem declarativa e tipada, cujos princípios estão baseados na lógica de primeira ordem, na teoria de conjuntos e em uma semântica operacional. Ela foi definida com o objetivo de ser uma linguagem de especificação menos intimidante, se comparada a linguagens formais tradicionais como, por exemplo, Z (WOODCOCK e DAVIES, 1996) e VDM-SL (JONES, 1989), por utilizar construções semelhantes àquelas encontradas em programas orientados a objetos, ao invés de empregar notações matemáticas (WARMER e KLEPPE, 2003). Entretanto, a elaboração de uma especificação em OCL que capture corretamente o universo que está sendo representado, e que, ao mesmo tempo, não apresente construções que comprometam o seu entendimento e a sua evolução, não é uma tarefa trivial (CORREA e WERNER, 2004a).

A constante evolução é uma característica intrínseca do desenvolvimento de software no mundo atual. Dentre os diversos fatores que podem contribuir negativamente para a evolução de um software, destaca-se a deterioração da sua estrutura interna, usualmente resultado do emprego de construções inadequadas que aumentam a complexidade, e dificultam o entendimento e a evolução de um software (MENS e TOURWÉ, 2004). Nesse cenário de constante evolução, as técnicas de reestruturação visam reduzir a complexidade do software através da melhoria incremental de sua estrutura interna (ARNOLD, 1989), (CHIKOFSKY e CROSS, 1990).

No contexto de desenvolvimento de software orientado a objetos, as técnicas e ferramentas de reestruturação têm sido aplicadas, fundamentalmente, no código dos

sistemas. Em (OPDYKE, 1992), foi definido, de forma pioneira, um conjunto de reestruturações aplicáveis na implementação de software orientado a objetos, com o objetivo de facilitar futuras adaptações e extensões. Posteriormente, as técnicas de reestruturação passaram a ser empregadas também em artefatos mais abstratos, tais como modelos UML. Entretanto, o emprego dessas técnicas sobre as restrições de um modelo especificadas em OCL, por exemplo, ainda não tinha sido explorado por nenhum trabalho do nosso conhecimento.

Ainda que não contenham construções que possam comprometer o seu entendimento e a sua evolução, modelos com restrições especificadas em OCL precisam ser avaliados em relação a potenciais problemas como, por exemplo, inconsistências, violações sintáticas, utilização incorreta de tipos, ou ainda, captura incorreta de fatos sobre o universo que está sendo representado. Além disso, ao reestruturarmos um modelo, devemos avaliar se as modificações efetuadas não introduziram problemas dessa natureza. Diferentes técnicas podem ser utilizadas para efetuar essas avaliações, podendo englobar análises tanto estáticas como dinâmicas (MALDONADO e FABBRI, 2001).

Análises dinâmicas envolvem a execução ou simulação da execução do modelo, com o objetivo de aferir a presença de certas propriedades, como, por exemplo, se as respostas de uma operação para determinados estímulos de entrada geram os efeitos esperados. Simulação por *snapshots* (LIU, 2004), animação (KAZMIERCZAK et al., 2000), modelos executáveis (MELLOR e BALCER, 2002) e prototipação (SOMMERVILLE, 1992) são exemplos de técnicas de análise dinâmica que podem ser aplicadas a modelos de software. Existem algumas abordagens voltadas para a elaboração de modelos UML executáveis, onde as ações são especificadas de forma imperativa, seja diretamente em uma linguagem de programação como, por exemplo, em (HAREL e GERY, 1997) e (SCHÄFER et al., 2001), seja através de uma linguagem de ação mais abstrata baseada na semântica de ações definida na especificação da UML 1.5, como em (MELLOR, 1998) e (KENNEDY CARTER LTD., 2002). Nenhuma dessas abordagens, entretanto, oferece recursos para a avaliação de restrições ou de especificações de operações produzidas com a OCL.

Em sistemas desenvolvidos de forma evolutiva, a presença de testes automatizados permite a aplicação da técnica de teste de regressão de forma eficiente (FEWSTER e GRAHAM, 1999). Teste de regressão visa verificar se uma nova versão

de um software continua realizando as mesmas funções, e gerando os mesmos resultados, como em sua versão anterior, através da reaplicação de casos de teste (PFLEEGER, 2001). Como uma reestruturação é uma operação que visa apenas modificar a estrutura de um software, preservando a sua semântica, espera-se que ele continue realizando as mesmas funções, e gerando os mesmos resultados, como em sua versão anterior. Portanto, teste de regressão automatizado pode ser utilizado para apoiar a verificação da preservação da semântica após uma reestruturação. Como nem todas as reestruturações podem ser automatizadas ou estar disponíveis em um ambiente de desenvolvimento, uma condição importante para a aplicação de reestruturações é a presença de testes automatizados sólidos (FOWLER, 1999).

1.2 Objetivos

Esta tese tem, portanto, os seguintes objetivos:

- Identificar e catalogar alguns tipos de construção que podem prejudicar o entendimento e a evolução de especificações de restrições de modelos elaboradas em OCL. O registro de um conjunto de tais construções pode ser utilizado não apenas como referência para a identificação de possíveis alvos de reestruturação em modelos já existentes, como também para evitar que essas construções sejam inseridas em novos modelos.
- Definir um conjunto de reestruturações que possam ser aplicadas de forma a remover construções potencialmente problemáticas de especificações elaboradas em OCL.
- Definir uma abordagem de apoio para a automação de reestruturações aplicadas a modelos e restrições em OCL.
- Construir o protótipo de uma ferramenta que ofereça apoio à realização de algumas atividades de verificação e validação de restrições especificadas em OCL, bem como às reestruturações, através da execução automática de testes de regressão.

1.3 Organização do Texto

Esta tese está organizada em nove capítulos, que incluem este capítulo

introdutório. No capítulo 2, são fornecidos os conceitos que proporcionam o embasamento teórico necessário à compreensão do trabalho aqui descrito, bem como das questões que foram alvo do estudo realizado. Os trabalhos relacionados a essas questões também são descritos neste capítulo.

Os capítulo 3 e 4 apresentam um conjunto de construções problemáticas que são frequentemente encontradas em especificações produzidas com a OCL, e o conjunto de reestruturações propostas para removê-las. O capítulo 3 apresenta as construções problemáticas e as respectivas reestruturações que envolvem apenas modificações em expressões OCL. O capítulo 4 apresenta construções e reestruturações que envolvem modificações não apenas nas expressões OCL, como também no modelo associado. Essas construções problemáticas foram identificadas a partir de um estudo realizado em diversas especificações como, por exemplo, as diversas versões da especificação da UML, artigos publicados em conferências internacionais, além de modelos elaborados por alunos de um curso de pós-graduação em desenvolvimento de software do NCE-UFRJ.

O capítulo 5 apresenta um estudo experimental que visou avaliar a viabilidade da aplicação das reestruturações propostas. Mais especificamente, fizemos uma avaliação inicial da influência exercida pelas reestruturações na compreensão de restrições de modelos especificadas em OCL.

O capítulo 6 descreve a abordagem proposta para apoiar a automação de reestruturações e a verificação da preservação da semântica do modelo após a realização das reestruturações. A automação de reestruturações aplicadas a modelos com restrições expressas em OCL é baseada na definição de operações de transformação, onde as condições para a aplicação de uma reestruturação, bem como os efeitos esperados, são especificadas em OCL, enquanto que as ações de transformação são definidas em uma linguagem de ações definida nesta tese, a OCL-AL (OCL Action Language), que estende a OCL de modo a permitir a elaboração de especificações executáveis de operações. A verificação da preservação da semântica do modelo após a realização de reestruturações consiste em empregar testes de regressão executados de forma automatizada, utilizando os casos de teste definidos para apoiar a avaliação da semântica do modelo. Embora, idealmente, esta verificação devesse ser garantida pela prova formal de equivalência das duas versões do modelo (antes e depois da reestruturação), nossa estratégia inicial de automação foi aceitar um nível de garantia

menor oferecido pelos testes, em troca de um resultado mais rápido e de uma menor dependência em relação a conhecimentos de provas formais por parte dos usuários.

O capítulo 7 descreve um conjunto de extensões à OCL que julgamos necessárias para a implementação e utilização prática da abordagem proposta. Neste capítulo, são descritas extensões que possibilitam a definição do escopo das modificações admissíveis por uma operação, correções na definição de operações da biblioteca padrão, modificações na definição de restrições associadas a hieraquias de classes, além da descrição detalhada da linguagem OCL-AL.

O capítulo 8 apresenta o software Odyssey-PSW, que foi desenvolvido com o objetivo de apoiar a edição, compilação, verificação de tipos e validação de restrições especificadas em OCL através da definição e execução de casos de teste.

O capítulo 9 apresenta as conclusões desta tese, com um resumo do trabalho efetuado, as suas contribuições, as limitações das propostas apresentadas e sugestões de trabalhos futuros.

Capítulo 2

Conceitos e Trabalhos Relacionados

2.1 Introdução

As principais questões abordadas por esta tese, bem como os principais conceitos e trabalhos relacionados, são apresentadas neste capítulo, que está estruturado da seguinte forma: a seção 2.2 apresenta uma breve introdução à estrutura utilizada pelo OMG para a definição de modelos e metamodelos, que é empregada na definição dos metamodelos da UML e da OCL. A seção 2.3 descreve as principais características da OCL. A seção 2.4 descreve questões ligadas à complexidade e manutenibilidade de modelos UML/OCL. A seção 2.5 apresenta os principais conceitos relacionados à reestruturação de modelos, e discute como esses conceitos podem ser utilizados de forma a minimizar os efeitos negativos produzidos por construções inadequadas. A seção 2.6 apresenta conceitos relacionados à verificação e validação de especificações, e discute como eles podem ser aplicados a modelos UML/OCL. Em particular, essa seção descreve as principais técnicas de análise dinâmica de especificações e os principais problemas ligados à sua aplicação em modelos UML/OCL. A seção 2.7 enumera algumas questões relacionadas à semântica da OCL no contexto da aplicação das técnicas de análise dinâmica de modelos UML/OCL. Finalmente, a seção 2.8 apresenta as considerações finais sobre este capítulo.

2.2 Elaboração de Modelos e os padrões OMG

A elaboração de modelos envolve a aplicação de três princípios fundamentais ao desenvolvimento de sistemas complexos: decomposição, abstração e hierarquia (BOOCH, 1994). Através da decomposição, um sistema complexo pode ser dividido em elementos de menor complexidade (PARNAS, 1972). A abstração permite nos concentrarmos nos aspectos relevantes de um problema, em um determinado nível de generalização (ROSS et al., 1975). As abstrações podem ser organizadas em hierarquias

(PFLEEGER, 2001), possibilitando a representação explicita de propriedades comuns e distintas entre diferentes elementos (BOOCH, 1994)

As representações gráficas para a elaboração de modelos de software se tornaram populares a partir das décadas de 70 e 80, sendo empregadas por diversos métodos estruturados de análise e projeto (DEMARCO, 1978), (GANE e SARSON, 1979), (YOURDON, 1989), e, posteriormente na década de 90, pelas diversas abordagens de modelagem orientadas a objetos (COAD e YOURDON, 1991), (RUMBAUGH, 1990), (JACOBSON, 1992), (BOOCH, 1994).

No final da década de 90, após ser adotada como padrão pelo OMG, a UML passou a ser largamente utilizada por diversos métodos, técnicas e ferramentas de apoio à modelagem de sistemas, no desenvolvimento de software em diversas áreas, tais como: comércio eletrônico, jogos, automação comercial e bancária, telecomunicações, robótica, aviação, dentre outras (BOOCH, 1999). A UML é uma linguagem que pode ser utilizada na especificação, visualização, construção e documentação de artefatos de sistemas de software, de negócio e de outros sistemas (OMG, 1999).

A UML define um conjunto de notações gráficas que podem ser utilizadas para descrever diversos aspectos de um sistema. Essas notações permitem a elaboração de diferentes tipos de diagramas como, por exemplo: diagramas de classes, de objetos, de casos de uso, de atividades, de seqüência, de colaboração, de estados, de implementação e de implantação. Através desse conjunto de diagramas, é possível representar aspectos estruturais e dinâmicos de um sistema.

A sintaxe abstrata da UML é definida segundo uma abordagem baseada em metamodelos. A Figura 2.1 apresenta a estrutura geral da arquitetura empregada nessa abordagem. O nível M3 corresponde ao MOF (*Meta Object Facility*). O MOF é um padrão, também adotado pelo OMG (OMG, 2002a), que define uma linguagem abstrata e uma estrutura para a especificação, construção e gerência de metamodelos, de forma independente de tecnologia. As construções definidas no MOF seguem o paradigma de orientação a objetos, compartilhando um conjunto de elementos com a UML como, por exemplo, classes, atributos, operações e associações.

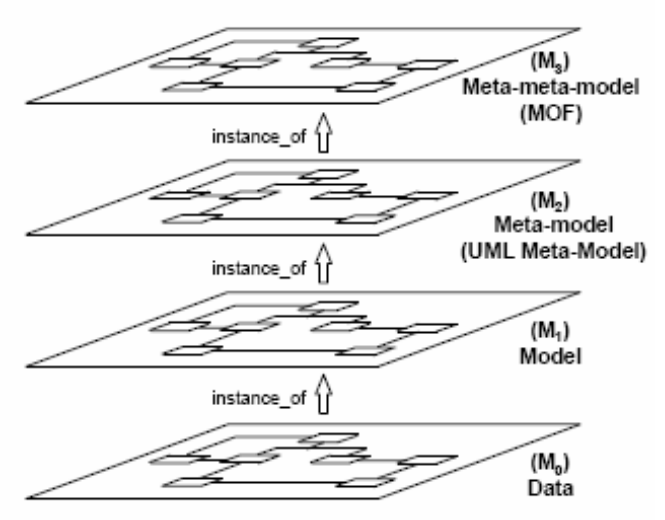


Figura 2.1– Estrutura de metamodelos da OMG

O nível M2 corresponde aos metamodelos definidos a partir do MOF, ou seja, os elementos de um metamodelo M2 são instâncias de elementos definidos no MOF (M3). O metamodelo da UML é um exemplo de metamodelo M2.

O nível M1 corresponde aos modelos que são gerados a partir dos metamodelos M2. A principal responsabilidade dos modelos neste nível é permitir a elaboração de representações de diferentes domínios como, por exemplo, software ou processos de negócio. O modelo de um software de gestão de uma locadora de vídeos é um exemplo de um modelo M1.

Um modelo M1 define o que deve acontecer quando seus elementos são instanciados no nível M0. Em linhas gerais, o nível M0 corresponde ao espaço de objetos e de ligações entre objetos resultantes da instanciação das classes e associações definidas em um modelo M1.

A Figura 2.2 apresenta um exemplo onde esses diferentes níveis são empregados. No nível M3, é definido um elemento denominado *Class*. Esse elemento é instanciado na definição do metamodelo da UML (nível M2), dando origem a dois elementos básicos: *Attribute* e *Class.* No nível M1, que corresponde ao modelo de uma locadora de vídeos, a classe *Vídeo* é uma instância do elemento *Class*, definido em M2, enquanto que o seu atributo *title* corresponde a uma instância do elemento *Attribute*. Os elementos do nível M0 correspondem a objetos da classe *Vídeo,* criados durante a execução do sistema.

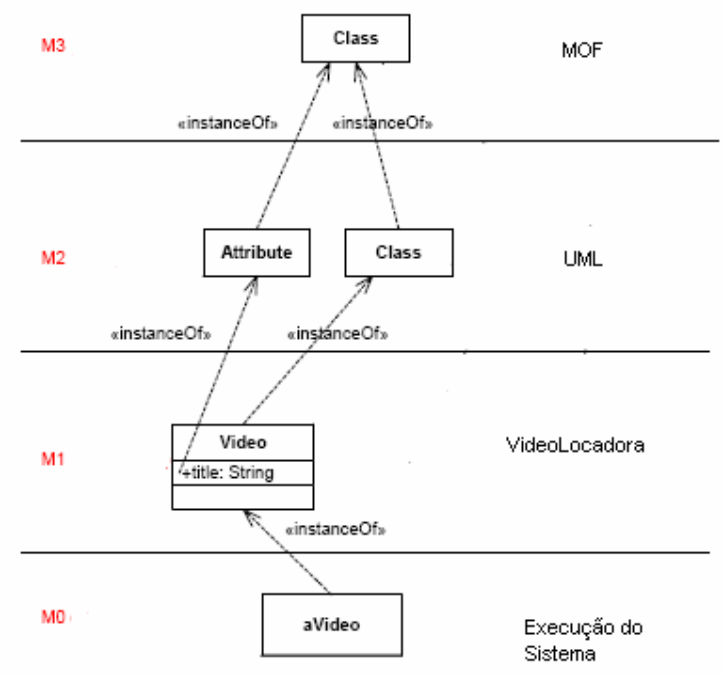


Figura 2.2 – Exemplo de modelos e metamodelos

2.3 OCL

Embora a UML ofereça um grande número de diagramas que possibilitam a construção de visões estáticas e dinâmicas de um sistema, eles não são suficientes para descrever todos os detalhes que compõem um modelo de software ou mesmo um metamodelo. Restrições, regras, definições contratuais de operações são alguns exemplos de informações que não são cobertas por esses diagramas, e que demandam uma especificação mais precisa (WARMER e KLEPPE, 2003).

Freqüentemente, essas definições são descritas em linguagem natural. Entretanto, especificações produzidas em linguagem natural estão intrinsecamente ligadas a problemas de ambigüidade (BERRY e KAMSTIES, 2004). O emprego de uma linguagem mais formal na produção dessas definições é uma alternativa natural para lidar com a questão da ambigüidade, que abre várias possibilidades de apoio automatizado ao longo do processo de desenvolvimento (PFLEEGER, 2001).

A OCL é uma linguagem de especificação textual e declarativa, também adotada como padrão pelo OMG (OMG, 2003a), que permite a definição precisa de restrições em modelos produzidos com a UML, bem como em metamodelos definidos a partir do MOF.

As restrições são especificadas em OCL através de expressões que não modificam o espaço de objetos onde elas são avaliadas. Na arquitetura de quatro níveis descrita anteriormente, esse espaço de objetos pode corresponder ao estado de um sistema (nível M0), a elementos de um modelo (nível M1), ou a elementos de um metamodelo (nível M2). Embora não possam produzir mudanças em um espaço de objetos, expressões OCL podem ser utilizadas para especificar as mudanças que devem ser produzidas por operações, através de pós-condições, por exemplo.

2.3.1 Tipos de Restrições e Expressões em OCL

As restrições e expressões especificadas em OCL devem ser associadas a elementos (classes, atributos, operações, associações) de um modelo. Dentre os diferentes tipos de definições que podem ser especificadas em OCL, destacam-se:

- ***Invariante (inv)***

  Corresponde a uma expressão que é associada a um classificador do modelo, indicando que o resultado da sua avaliação deve ser verdadeiro para todas as instâncias que sejam de um tipo compatível com esse classificador.

- ***Derivação de atributos e associações (derive)***

  Corresponde a uma expressão que define a regra de derivação do valor de um atributo ou de uma associação a partir de outros elementos do modelo.

- ***Corpo de operação de consulta (body)***

  Corresponde a uma expressão que especifica, de forma explícita, o resultado de uma operação de consulta definida em um classificador do modelo. Uma operação de consulta não pode produzir modificações no espaço de objetos onde ela seja executada.

- ***Valor inicial de atributos (init)***

  Corresponde a uma expressão que define o valor inicial de um atributo no momento em que for criado um objeto do classificador onde esse atributo tenha sido definido.

- ***Pré e pós-condições (pre e post)***

  Corresponde a expressões utilizadas para definir, de forma declarativa, a semântica das operações de um modelo. Na OCL, a definição de pré e pós-condições para as operações das classes de um modelo está associada ao princípio de Projeto por Contrato[1] (MEYER, 1992), onde uma operação é responsável por produzir certos resultados (obrigações ou pós-condições), apenas se certas condições (direitos ou pré-condições) forem atendidas.

2.3.2 Tipos

A OCL é uma linguagem tipada. Toda expressão tem um tipo que determina o domínio do seu resultado e das operações que podem ser aplicadas. Quatro tipos primitivos são predefinidos pela linguagem: *Boolean*, *Integer*, *Real* e *String*.

Além dos tipos primitivos, os classificadores e as enumerações definidos no modelo também fazem parte dos tipos disponíveis para as expressões OCL. Expressões OCL podem resultar em um valor primitivo, um objeto, uma tupla, ou uma coleção desses elementos. A OCL define quatro tipos de coleção (*Set, Bag, Sequence* e *OrderedSet*), que correspondem à combinação de duas propriedades: a possibilidade de ocorrência de repetição de elementos na coleção, e a existência de uma ordem entre os elementos da coleção. A Tabela 2.1 sintetiza a correspondência entre os tipos de coleção e essas duas propriedades.

Coleção	**Repetição de elementos**	**Ordem dos Elementos**
Set	Não	Não
Bag	Sim	Não
Sequence	Sim	Sim
OrderedSet	Não	Sim

Tabela 2.1 - Tipos de coleção na OCL

A OCL define um conjunto de operações que permite a manipulação de valores dos tipos primitivos e também das coleções como, por exemplo, operações lógicas (*and, or, xor, implies, if-then-else-endif*), aritiméticas (*+, -, *, /*), manipulação de strings (*size,*

[1] *Design By Contract*

concat, substring), manipulação de coleções (*size, includes, isEmpty*). A relação completa dos tipos e das suas respectivas operações é descrita na especificação da linguagem (OMG, 2003a). As operações de consulta definidas nos classificadores do modelo também podem ser utilizadas nas expressões.

2.3.3 Navegação

A OCL é uma linguagem navegacional, ou seja, a partir de um elemento inicial, é possível navegar pelas associações definidas no modelo. O elemento inicial de uma navegação pode corresponder a um objeto ou a uma coleção de objetos. Cada navegação resulta em uma coleção contendo os objetos associados ao elemento inicial. O tipo da coleção resultante é definido de acordo com a multiplicidade e com o classificador destino da navegação. A navegação por uma associação é definida por uma expressão com a estrutura <*origem*>.<*papel*>, onde *origem* corresponde a um objeto ou a uma coleção de objetos de uma classe *A*, e *papel* corresponde ao nome do papel (*rolename*) de uma classe associada à classe *A*.

Na expressão *self.contas->select(conta | conta.tipoOuro())->size(), por exemplo,* a expressão *self.contas* resulta na coleção de objetos da classe *ContaCorrente* correspondente à navegação pela associação entre as classes *Cliente* e *ContaCorrente*, definida no diagrama da Figura 2.3. Em função da multiplicidade de *ContaCorrente* nessa associação, o resultado da expressão é do tipo *Set(ContaCorrente)*, correspondendo a todas as instâncias de *ContaCorrente* associadas a um objeto *Cliente (self)*. A expressão *self.contas.fundos*, por sua vez, resulta em uma coleção do tipo *Bag(FundoInvestimento)*, correspondendo a todos os fundos de investimento associados a todas as contas correntes de um cliente.

As chamadas às operações de manipulação de coleção devem ser precedidas do operador ->. Nesse mesmo exemplo, a expressão *self.contas->select(conta | conta.tipoOuro())->size()* corresponde à chamada da operação *select* a partir da coleção resultante da expressão *self.contas*. Essa expressão resulta em um subconjunto da coleção *self.contas*, a partir do qual, a operação *size é chamada*, resultando no número de elementos desse subconjunto.

2.3.4 Valor Indefinido

Algumas expressões, quando avaliadas, podem resultar em um valor indefinido. A divisão de um número por zero e o resultado da operação *first* aplicada a uma coleção sem elementos são alguns exemplos de expressões que resultam em um valor indefinido. Além disso, como um atributo de uma classe pode ser definido com multiplicidade [0..1], indicando que nem toda instância dessa classe precisa ter um valor definido para esse atributo, expressões que envolvam um atributo com essa multiplicidade também podem resultar em um valor indefinido.

Para lidar com esse fato, as operações lógicas envolvendo expressões OCL são definidas considerando três valores: verdadeiro, falso e indefinido (⊥). A Tabela 2.2 apresenta a semântica das operações do tipo *Boolean* considerando esses três possíveis valores.

b_1	b_2	b_1 and b_2	b_1 or b_2	b_1 xor b_2	b_1 implies b_2	not b_1
false	false	false	false	false	true	true
false	true	false	true	true	true	true
true	false	false	true	true	false	false
true	true	true	true	false	true	false
false	⊥	false	⊥	⊥	true	true
true	⊥	⊥	true	⊥	⊥	false
⊥	false	false	⊥	⊥	⊥	⊥
⊥	true	⊥	true	⊥	true	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥

Tabela 2.2 – Semântica das operações do tipo Boolean

Em geral, o resultado de expressões envolvendo um valor indefinido é também indefinido, com exceção de algumas expressões envolvendo operações booleanas, ou de algumas operações de coleção como, por exemplo, *select, forAll, exists.* A operação booleana *oclIsUndefined*() permite verificar se o valor de uma expressão é indefinido.

2.3.5 Exemplos de Definições em OCL

Essa seção apresenta alguns exemplos de definições especificadas em OCL, produzidas a partir dos elementos definidos pelo diagrama de classes apresentado na Figura 2.3. De acordo com esse diagrama, um cliente pode possuir uma ou mais contas correntes. A uma conta corrente, podem estar vinculados zero ou mais fundos de investimento. A classe Conta é uma generalização de conta corrente e fundo de investimento.

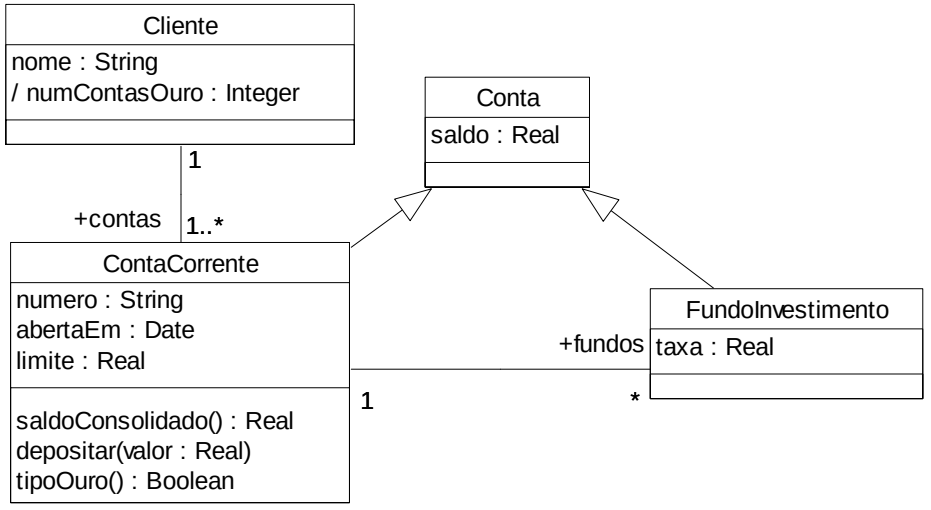


Figura 2.3 - Exemplo de um diagrama de classes para um sistema bancário

Uma restrição do tipo invariante é definida no contexto de um classificador por uma expressão através da palavra reservada *inv*. A restrição presente nas linhas 1-2 da Figura 2.4 se aplica a todas as instâncias de *ContaCorrente*, enquanto que a restrição especificada nas linhas 4-5 se aplica a todas as instâncias do tipo *Conta* que, neste exemplo, correspondem às instâncias das classes *ContaCorrente* e *FundoInvestimento*.

```
1 context ContaCorrente
2 inv limiteMinimo: limite > 100
3
4 context Conta
5 inv: saldo >= 0
```

Figura 2.4 – Exemplo de uma restrição do tipo invariante

Um exemplo de definição da expressão associada a um atributo derivado é apresentado na Figura 2.5. Nesse exemplo, a expressão associada ao atributo derivado *numContasOuro* é definida pelo número de elementos do subconjunto das contas do cliente, tais que a avaliação da operação de consulta *tipoOuro* resulte no valor verdadeiro. A palavra reservada *self* é utilizada para referenciar a instância correspondente ao elemento que determina o contexto da expressão (*Cliente*), isto é, a instância a partir da qual a expressão será avaliada.

```
1 context Cliente::numContasOuro : Integer
2 derive: self.contas->select(conta | conta.tipoOuro())->size()
```

Figura 2.5 – Exemplo de expressão associada a atributos derivados

O resultado de uma operação de consulta pode ser definido por uma expressão OCL associada a essa operação através da palavra reservada *body*. A Figura 2.6 apresenta a definição da operação de consulta *tipoOuro*. De acordo com essa definição, uma conta corrente é considerada *Ouro,* quando o seu saldo consolidado for maior que 15000.

```
1 context Conta::tipoOuro() : Boolean
2 body: self.saldoConsolidado() > 15000
```

Figura 2.6 – Exemplo de definição do corpo de uma operação de consulta

O valor inicial para um atributo pode ser definido através de uma expressão OCL associada ao atributo pela palavra reservada *init.* No exemplo da Figura 2.7, o valor inicial do atributo taxa para as instâncias da classe *FundoInvestimento* é definido pela expressão 2.5.

```
1 context FundoInvestimento::taxa : Real
2 init: 2.5
```

Figura 2.7 – Exemplo de expressão de inicialização de um atributo

A semântica de uma operação é especificada através de pré e pós-condições, conforme ilustrado pela Figura 2.8. Neste exemplo, duas pré-condições foram definidas para a operação *depositar* da classe *ContaCorrente*. Um nome pode ser associado a cada pré ou pós-condição (por exemplo, *valorMinimo*, *valorMaximo* e *saldoAtualizado*). Como a avaliação de uma pós-condição considera dois espaços de objetos (e_{antes} e $e_{depois,}$, onde e_{antes} corresponde ao momento da chamada, e e_{depois} corresponde ao momento após a execução da operação), expressões sucedidas por *@pre* indicam que elas devem ser avaliadas no espaço e_{antes}, enquanto que as demais expressões são avaliadas no espaço e_{depois}. Assim, a pós-condição deste exemplo indica que, se as pré-condições forem verdadeiras, o saldo da conta após a execução da operação *depositar* deverá ser igual à soma do parâmetro *valor* e do *saldo* da conta no momento da chamada dessa operação.

```
1 context ContaCorrente::depositar(valor : Real)
2 pre valorMinimo: valor > 0
3 pre valorMaximo: valor < 1000
4 post saldoAtualizado: saldo = saldo@pre + valor
```

Figura 2.8 – Exemplo de especificação de pré e pós-condições

2.4 Construções Problemáticas em Especificações OCL

Ao invés de empregar notações matemáticas, a OCL utiliza construções sintáticas semelhantes àquelas encontradas em linguagens de programação orientada a objetos, com o objetivo de tornar a linguagem menos intimidante (WARMER e KLEPPE, 2003), se comparada com as linguagens formais tradicionais como, por exemplo, Z (WOODCOCK e DAVIES, 1996) e VDM-SL (JONES, 1989). Isso não impede, entretanto, que uma especificação OCL apresente problemas de legibilidade e manutenibilidade, especialmente em casos onde expressões inadequadas sejam empregadas, ou ainda, pela ausência de construções mais genéricas no modelo.

Para ilustrar esse problema, a Figura 2.9 apresenta um exemplo de uma restrição, extraída da especificação da UML 1.3 (OMG, 1999), que corresponde a uma regra associada à classe *Collaboration.* A parte relevante do modelo é apresentada no diagrama da Figura 2.10.

[3] Se um *ClassifierRole* ou um *AssociationRole* não tiver um nome, então ele deve ser o único associado à sua base.

```
self.allContents->forAll ( p |
  (p.oclIsKindOf (ClassifierRole) implies
      p.name = '' implies
          self.allContents->forAll ( q |
              q.oclIsKindOf(ClassifierRole) implies
                  (p.oclAsType(ClassifierRole).base =
                      q.oclAsType(ClassifierRole).base implies
                          p = q) ) )
  and
  (p.oclIsKindOf (AssociationRole) implies
      p.name = '' implies
          self.allContents->forAll ( q |
              q.oclIsKindOf(AssociationRole) implies
                  (p.oclAsType(AssociationRole).base =
                      q.oclAsType(AssociationRole).base implies
                          p = q) ) )
)
```

Figura 2.9 – Exemplo de uma restrição complexa em OCL

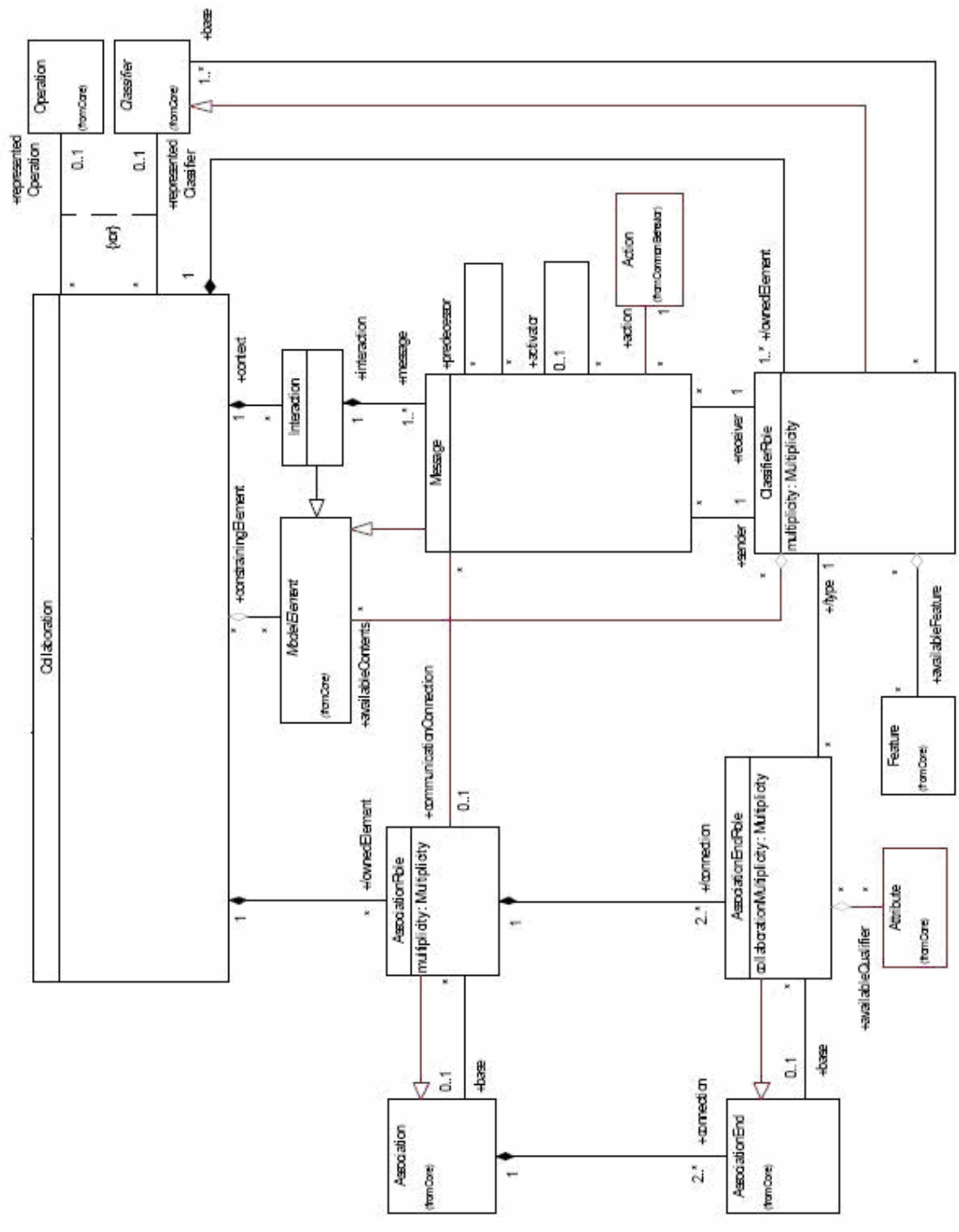


Figura 2.10 – Fragmento do metamodelo da UML 1.3

Nesse exemplo, a expressão *p.oclIsKindOf(ClassifierRole)* resultará no valor verdadeiro, se o tipo do objeto *p* for *ClassifierRole* ou algum subtipo de *ClassifierRole.*

A operação *oclAsType* possibilita a coerção de tipos[2], e é tipicamente utilizada para acessar propriedades específicas definidas em um subtipo, a partir de uma referência para um tipo ancestral. Na expressão *p.oclAsType(ClassifierRole).base*, por exemplo, *p* é um iterador do tipo *ModelElement,* e como a propriedade *base* está definida em *ClassifierRole*, que é uma especialização de *ModelElement,* a coerção de *ModelElement* para *ClassifierRole* deve ser efetuada através da expressão *p.oclAsType(ClassifierRole),* para que a propriedade *base* possa ser acessada a partir de *p*.

Em uma primeira análise da restrição apresentada na Figura 2.9, a sua complexidade pode ser interpretada como resultado das características da OCL que, em muitos casos, levam à utilização de operações como *oclIsKindOf* e *oclAsType (VAZIRI e JACKSON, 2002)*. Uma análise mais detalhada, entretanto, revela que sua complexidade é resultado da utilização de construções inadequadas, da ausência de uma generalização na definição das classes do modelo, bem como do emprego de nomes inadequados para as variáveis presentes na expressão.

A restrição original consiste na aplicação de um quantificador universal (*forAll)* sobre todos os elementos associados à classe *Collaboration (self.allContents*). Uma vez que a expressão associada ao quantificador universal apresenta a estrutura (*p.oclIsKindOf(ClassifierRole) implies X and p.oclIsKindOf(AssociationRole) implies Y*), apenas os elementos do tipo *ClassifierRole* e *AssociationRole* são relevantes nessa restrição. Portanto, o primeiro problema dessa expressão corresponde à utilização da expressão *self.AllContents* em lugar das navegações específicas *self.associationRole* e *self.classifierRole*.

Além disso, a única diferença entre as expressões *X* e *Y* reside no nome de uma classe (*ClassifierRole* em *X* e *AssociationRole* em *Y)*. Em geral, a presença de construções OCL duplicadas é um indicador de uma possível ausência de generalização no modelo. A especificação desse exemplo pode ser simplificada, se introduzirmos um conceito (*Role*) correspondente à generalização das classes *ClassifierRole*, *AssociationRole* e *AssociationEndRole*, conforme o diagrama apresentado na Figura 2.11. Todo elemento *Role* possui uma operação *base* que retorna o conjunto de elementos da associação específica de uma subclasse de *Role* com o(s) seus(s)

[2] type casting

elemento(s) base. Dessa forma, a restrição poderia passar a ser expressa pelo conjunto de definições apresentadas na Figura 2.12.

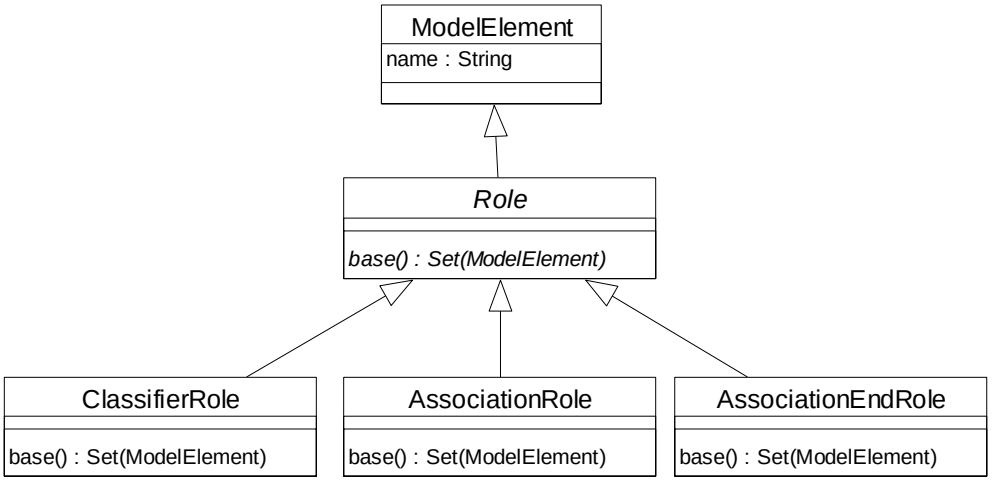


Figura 2.11 – Versão reestruturada do metamodelo da UML 1.3

```
context Collaboration
-- allRoles é uma definição que corresponde ao conjunto de todos os
-- AssociationRoles e ClassifierRoles associados a uma Collaboration
def: allRoles : Collection(Role) =
        allAssociationRoles->union(allClassifierRoles)
def: allAssociationRoles : Collection(Role) = self.associationRole
def: allClassifierRoles  : Collection(Role) = self.classifierRole

-- allRolesWithBlankName é um subconjunto de allRoles, considerando
-- apenas os elementos que não tenham nome
def: allRolesWithBlankName : Collection(Role) =
     self.allRoles->select(role | role.name = '')

-- Em uma Collaboration, todo ClassifierRole ou AssociationRole que não tiver um
-- nome deve ser o único associado à sua base.
inv: self.allRolesWithBlankName->forAll(role1 |
          self.allRoles->forAll(role2 |
               role1.base() = role2.base() implies role1 = role2))
```

Figura 2.12 – Versão reestruturada de uma restrição da UML 1.3

Assim como esse exemplo, existem diversas outras situações onde uma especificação de restrições em OCL pode apresentar problemas como complexidade

desnecessária e estruturação inadequada, ou ainda, revelar possibilidades de modificações no modelo. Esta tese apresenta um conjunto de *OCL smells* que são frequentemente encontrados em especificações produzidas com a OCL. Este conjunto resultou da análise de especificações produzidas por alunos de um curso de pós-graduação lato sensu da UFRJ, das regras de boa formação de modelos definidas nas especificações da UML 1.x, UML 2.0 (superestrutura e infra-estrutura), MOF 1.4 e OCL 2.0, bem como de artigos científicos internacionais contendo restrições expressas em OCL, tais como (CARIOU et al., 2004), (FERNANDEZ-MEDINA e PIATTINI, 2004) e (ACKERMANN, 2005).

A identificação e o registro de construções inadequadas estão diretamente relacionados com os conceitos de *code smell* e anti-padrão. O termo *Code smell* refere-se a certas estruturas presentes no código de um software que sugerem a necessidade de uma reestruturação (FOWLER, 1999). Duplicação de código, implementação de métodos grandes e complexos, encadeamento de mensagens são alguns exemplos de *code smells*. Um anti-padrão descreve um problema recorrente que traz conseqüências negativas para o projeto. O objetivo dos anti-padrões é categorizar, nomear e descrever soluções ruins que ocorrem repetidamente, além de mostrar possíveis alternativas melhores que poderiam ter sido empregadas (BROWN et al., 1998).

Existem diversos trabalhos ligados à identificação e ao registro de soluções problemáticas em sistemas orientados a objetos, tais como (DUCASSE et al., 1999), (BALAZINSKA et al., 2000), (CORREA et al., 2000), (KATAOKA et al., 2001), (SIMON et al., 2001), (EMDEN e MOONEN, 2002) e (TOURWÉ e MENS, 2003). Entretanto, o foco desses trabalhos reside na identificação de problemas no código ou no projeto detalhado de software, não abordando especificações produzidas com a OCL.

2.5 Reestruturação

Construções problemáticas em uma especificação OCL podem ser substituídas por outras mais adequadas através de modificações efetuadas, não apenas nas expressões OCL, como também no modelo. Uma outra questão relevante neste contexto é que as mudanças efetuadas nos elementos do modelo devem considerar um eventual impacto nas restrições expressas em OCL. A eliminação de uma associação entre duas classes, por exemplo, afeta todas as restrições que contenham expressões de navegação

envolvendo essa associação.

Em geral, as modificações realizadas em artefatos de um software podem ser classificadas em duas grandes categorias:

- modificações que não alteram a semântica funcional dos artefatos, isto é, modificações que são efetuadas na estrutura de um software, preservando suas funcionalidades, com o objetivo de facilitar a sua futura evolução. Essa categoria de mudança é conhecida como reestruturação (GRIWSWOLD, 1991).
- modificações que alteram a semântica dos artefatos, normalmente associadas a manutenções corretivas, adaptativas ou evolutivas do software (PRESSMAN, 2004).

O termo reestruturação pode ser definido como uma transformação de um artefato de uma forma de representação para outra, no mesmo nível relativo de abstração, que preserva a semântica do artefato transformado (CHIKOFSKY e CROSS, 1990). Reestruturação é uma técnica que desempenha um importante papel no contexto de evolução de software, e tem como principal objetivo melhorar a estrutura de um software em relação a certas características de qualidade, de forma a evitar que ela fique complexa a ponto de exigir um esforço cada vez maior em evoluções subseqüentes (MENS e TOURWÉ, 2004). Extensibilidade, modularidade, reusabilidade, manutenibilidade, eficiência são alguns exemplos de características de qualidade que estão associadas à aplicação de reestruturações.

No contexto de desenvolvimento de software orientado a objetos, as técnicas e ferramentas têm sido aplicadas, fundamentalmente, na reestruturação de código. Em (OPDYKE, 1992), um conjunto de reestruturações aplicáveis na implementação de software orientado a objetos foi proposto com o objetivo de facilitar futuras adaptações e extensões. Essas reestruturações correspondem a operações primitivas como, por exemplo, adicionar um elemento (classe, atributo ou operação); remover um elemento; mover um atributo ou operação para outra classe. Uma proposta de formalização dessas reestruturações, baseada em pré e pós-condições, e para a sua automação em programas Smalltalk foi descrita em (ROBERTS, 1999). Uma avaliação do impacto da utilização dessas reestruturações na evolução de projetos de software pode ser encontrada em (TOKUDA e BATORY, 2001). Em (CORNÉLIO et al., 2002), é descrita uma

abordagem formal de reestruturação de programas orientados a objetos através de refinamentos. Atualmente existem vários catálogos de reestruturações aplicáveis a programas orientados a objetos como, por exemplo, o catálogo proposto em (FOWLER, 1999) para código Java, sendo que vários ambientes de codificação em Java oferecem apoio automatizado para algumas dessas reestruturações.

Nos últimos anos, as técnicas de reestruturação passaram a ser aplicadas em artefatos mais abstratos. Em (EVANS, 1998), são definidas regras de transformação formais aplicáveis a diagramas de classes. Em (SUNYÉ et al., 2001), são descritas algumas reestruturações aplicáveis a diagramas de classes e a diagramas de estados de um modelo UML, além de alguns aspectos relativos à preservação da integridade dos diagramas após essas reestruturações. Em (BOGER et al., 2002), são definidas algumas reestruturações aplicáveis em diagramas de classes, as quais são basicamente adaptações de reestruturações definidas em (OPDYKE, 1992), e em diagramas de estados e de atividades. Em (PORRES, 2003), são propostas algumas reestruturações que podem ser aplicadas em diagramas UML através de operações de transformação de modelos, implementadas com uma linguagem de script baseada em Phyton, denominada SMW. Em (GORP et al., 2003), é descrita uma extensão ao metamodelo da UML, que visa apoiar operações de reestruturação aplicadas a modelos, de forma que seja possível manter a consistência do modelo com o código fonte já existente.

Alguns trabalhos mais recentes que abordam a reestruturação de modelos UML levam em consideração a possível existência de restrições especificadas em OCL como parte do modelo (GHEYI et al., 2005), (MASSONI et al., 2005). (RAMOS et al., 2005). Entretanto, nenhum desses trabalhos relacionados aborda a definição de reestruturações aplicáveis a um modelo com o objetivo de remover expressões OCL complexas ou inadequadas (CORREA e WERNER, 2004a).

A definição de reestruturações aplicáveis a um modelo contendo restrições expressas em OCL é particularmente relevante no contexto da elaboração de modelos que façam uso intensivo da OCL. A especificação da UML 2.0 (OMG, 2004b), por exemplo, faz uso intensivo da OCL na definição de regras para modelos bem formados, mas apresenta uma série de construções desnecessariamente complexas que comprometem o seu entendimento e a sua manutenção. Esta tese descreve um conjunto de reestruturações que podem ser aplicadas com o objetivo de reduzir a complexidade das restrições que façam parte de um modelo.

2.6 Verificação e Validação de Modelos

Os artefatos produzidos ao longo de um processo de desenvolvimento de software compreendem um conjunto de informações em diferentes níveis de abstração e um conjunto de transformações e decisões associadas a essas transformações. Ao longo das atividades necessárias para obter, comunicar e transformar essas informações, existe a possibilidade dos artefatos gerados apresentarem enganos, interpretações errôneas, omissões e inconsistências, o que pode resultar em falhas no funcionamento do sistema resultante (MALDONADO e FABBRI, 2001). Uma vez que esses defeitos podem ser introduzidos em qualquer instante do processo de desenvolvimento, é importante que eles sejam detectados o mais cedo possível, já que o custo relativo para a sua correção é maior, quanto mais tarde eles forem detectados (PRESSMAN, 2004).

No caso de modelos UML/OCL, a OCL pode ser utilizada para definir, de forma precisa e não ambígua, as restrições e a semântica das operações de um modelo. Entretanto, mesmo que um modelo não contenha ambigüidades e não apresente estruturas duplicadas ou desnecessariamente complexas, ele pode conter problemas como inconsistências, violações das regras da linguagem, utilização incorreta de tipos, ou ainda, não capturar de forma correta o universo que está sendo representado.

O objetivo da verificação é assegurar que o sistema, ou uma determinada parte do mesmo, esteja sendo construído corretamente, verificando-se, inclusive, se os métodos e processos de desenvolvimento foram adequadamente aplicados. A validação, por sua vez, tem o objetivo de assegurar que o sistema atende às reais necessidades dos usuários (PFLEEGER, 2001). Desta forma, as atividades de verificação visam responder à pergunta “estamos construindo o produto corretamente?”, enquanto que as atividades de validação visam responder à pergunta: “estamos construindo o produto correto?” (BOEHM, 1981).

As atividades de verificação e validação podem englobar análises estáticas e dinâmicas dos diversos artefatos produzidos. Essas análises podem ser realizadas de forma manual ou automática, dependendo da disponibilidade de ferramentas de apoio (MALDONADO e FABBRI, 2001). As diversas técnicas de verificação e validação devem ser vistas como atividades complementares, sendo que alguma combinação dessas técnicas se faz necessária para a produção de sistemas com qualidade (TRAVASSOS et al., 2001).

Análises estáticas englobam atividades que analisam a forma e a estrutura de um artefato, não dependendo da execução propriamente dita do produto. Revisão Técnica é uma das possíveis formas de análise estática, e representa uma das principais atividades de controle de qualidade em um processo de desenvolvimento de software. Existem várias formas de revisar os artefatos produzidos, destacando-se inspeções, walkthroughs e revisões individuais (HUMPHREY, 1995). Técnicas específicas para revisão de modelos UML são descritas em trabalhos como (TRAVASSOS et al., 1999) e (LAITENBERGER e ATKINSON, 1999).

Uma outra técnica de análise estática, utilizada especialmente em especificações formais, é a prova formal, através da qual podemos demonstrar matematicamente que determinadas propriedades acerca de uma especificação são verdadeiras ou falsas. Provas formais também podem ser utilizadas para demonstrar a correção do refinamento de um artefato como, por exemplo, para verificar se uma implementação é um refinamento correto de sua respectiva especificação, ou ainda, se um projeto detalhado é um refinamento correto de um projeto de alto nível (WOODCOCK e DAVIES, 1996). Entretanto, a aplicação de provas formais requer um bom conhecimento de teorias matemáticas e das estratégias de prova envolvidas (KAZMIERCZAK et al., 2000). Em relação a especificações OCL, uma abordagem para a verificação de programas implementados em Java Card em relação a um modelo UML/OCL, assim como para a verificação formal de alguns tipos de inconsistências em especificações OCL é proposta em (AHRENDT et al., 2005). Um ambiente interativo de prova para a OCL, baseado no provador de teoremas Isabelle (NIPKOW et al., 2002), é descrito em (BRUCKER e WOLFF, 2002).

Ainda no contexto de análises estáticas, a técnica de *model checking* pode ser utilizada para verificar propriedades em sistemas concorrentes. Especificações sobre o sistema são expressas em fórmulas utilizando lógica temporal, e algoritmos simbólicos eficientes são utilizados para percorrer o modelo e verificar se certas propriedades são satisfeitas pela especificação, gerando contra-exemplos, caso contrário (CLARKE et al., 1994).

Outras formas de análise estática que podem ser aplicadas em especificações produzidas com linguagens de sintaxe e semântica bem definidas, como a OCL, por exemplo, são a análise sintática e a verificação de tipos, que possibilitam verificar se uma especificação não possui defeitos referentes à utilização incorreta da linguagem. A

análise sintática e a verificação de tipos estão disponíveis em alguns protótipos acadêmicos como Dresden OCL compiler (LOECHER e OCKE, 2004), OCLE (CHIOREAN et al., 2004), KMF (AKEHURST e PATRASCOIU, 2004) e USE (RICHTERS e GOGOLLA, 2002), e, mais recentemente, em algumas ferramentas comerciais como IBM Rational Software Architect (www-306.ibm.com/software/awdtools/architect/swarchitect), Borland Together Architect (www.borland.com/products/together), Magic Draw (www.magicdraw.com) e Octopus (www.klasse.nl/octopus).

Análises dinâmicas, por outro lado, envolvem a execução ou simulação da execução do artefato, com o objetivo de aferir a presença de certas propriedades, como, por exemplo, se as respostas de uma operação para determinados estímulos de entrada geram os efeitos esperados. As técnicas dinâmicas não asseguram a correção de uma especificação. Apesar das técnicas de análise dinâmica serem capazes apenas de mostrar a presença de defeitos, e não de garantir a sua ausência (MILLER e STROOPER, 2001), existem vários relatos da efetividade de sua aplicação tanto em pequenos projetos acadêmicos (WEST e EAGLESTONE, 1992), como em projetos maiores na indústria (GRAVELL e HENDERSON, 1996), (BICARREGUI, 1997), (BERG et al., 1999), (FENKAM et al., 2002). Animação, prototipação, modelos executáveis e simulação por *snapshots,* são alguns exemplos de técnicas de análise dinâmica que podem ser aplicadas a modelos de software. Essas técnicas são brevemente descritas a seguir.

2.6.1 Animação

Animação é uma técnica de validação baseada na execução do modelo que é bastante explorada no contexto de especificações baseadas em linguagens formais como Z, por exemplo. Ao invés de provar formalmente que uma especificação apresenta certas propriedades, a animação permite que o especificador levante questões sobre a especificação e obtenha respostas de forma rápida e automática, através de um ambiente interativo de execução da especificação. Embora não assegure a correção da especificação, a técnica de animação oferece uma alternativa atraente e de menor custo às provas formais, principalmente nos estágios iniciais do desenvolvimento (KAZMIERCZAK et al., 2000).

A técnica de animação permite aos envolvidos, através da exploração de cenários concretos e da observação dos efeitos produzidos, aumentarem seu grau de confiança de

que a especificação reflete corretamente a sua intenção original. Desta forma, a possibilidade de animar uma especificação oferece uma combinação efetiva de formalismo e pragmatismo.

Diversos trabalhos na área de especificações formais resultaram em uma grande variedade de ferramentas ou protótipos para animação, especialmente para a linguagem Z. Dentre estas ferramentas, destacam-se:

- *PiZa* (Prolog Implemented Zed Animator) (HEWITT, 1997), que é uma ferramenta capaz de executar um subconjunto da linguagem de especificação Z, traduzindo diretamente especificações escritas em Z para a linguagem Prolog.
- *ZANS* (Z Animation System) (JIA, 1995) é um animador projetado para animar especificações escritas em um subconjunto da linguagem Z. *ZANS* é capaz de animar apenas esquemas de operação explícitos, isto é, aqueles onde as variáveis de saída podem ser determinadas direta ou indiretamente a partir das variáveis de entrada. Os esquemas implícitos, isto é, esquemas onde as variáveis de saída são restritas pelos valores de entrada, não são tratados pela ferramenta.
- *VENUS* (JIA, 1997) é uma evolução da ferramenta *ZANS* e é baseada na combinação de UML e Z, denominada AML (Augmented ObjectOriented Modeling Language). Na AML, o desenvolvedor especifica as classes e associações graficamente com a UML, sendo que os atributos e as operações de cada classe são especificados em Z. Os elementos produzidos em UML são traduzidos para Z, e a partir de então, o usuário pode usufruir dos recursos de animação oferecidos pelo ZANS. Entretanto, o resultado da animação é visualizado com uma interface textual e na sintaxe da linguagem Z.
- *POSSUM* (HAZEL et al., 1997; HAZEL et al., 1998) é uma ferramenta de animação de especificações produzidas com a linguagem Sum, uma extensão da linguagem Z. A ferramenta de animação executa a especificação diretamente, sem exigir qualquer modificação da especificação original. Além disso, o ambiente possui um ambiente gráfico e interativo que facilita a sua utilização.
- *IFAD VDM-SL/VDM++ Toolbox* (ELMSTROM et al., 1994) é um conjunto de ferramentas que fornece suporte para o desenvolvimento de especificações formais com o padrão ISO VDM-SL, e com a extensão orientada a objetos (VDM++). Uma das ferramentas do conjunto oferece uma ligação com o

software Rational Rose, possibilitando a tradução de diagramas UML para VDM ou VDM++. A especificação resultante pode ser, então, interpretada e depurada, desde que as especificações das operações sejam definidas, de forma explícita, em VDM++.

No contexto de especificações OCL, a ferramenta *USE* (UML-based Specification Environment) (RICHTERS e GOGOLLA, 2000) oferece apoio à validação de modelos de projeto UML/OCL através de recursos de animação. A definição das classes é efetuada em uma linguagem textual própria da ferramenta. Além da verificação sintática e de tipos, a ferramenta possui um módulo onde o usuário pode modificar o estado do sistema criando ou destruindo objetos, criando ou destruindo ligações entre objetos, ou ainda, mudando os valores de atributos de objetos. Essas modificações são realizadas de forma interativa através de comandos escritos em uma linguagem de script do ambiente.

A ferramenta *USE* permite, ainda, que essas modificações do estado do sistema possam ser feitas dentro de uma operação, simulando mudanças geradas pela sua execução. Ao final dessa execução simulada, todas as pós-condições da operação são verificadas. O USE informa o estado de cada pós-condição, e caso alguma não tenha sido atendida, o usuário pode solicitar a exibição do estado de certos objetos do sistema, de modo a facilitar a investigação dos motivos que causaram a falha.

A seguir apresentamos um pequeno exemplo para ilustrar a filosofia da *USE*. A Figura 2.13 apresenta um diagrama de classes representando um modelo de empregados (Employee). Este modelo deve ser traduzido para a especificação textual ilustrada na Figura 2.14.

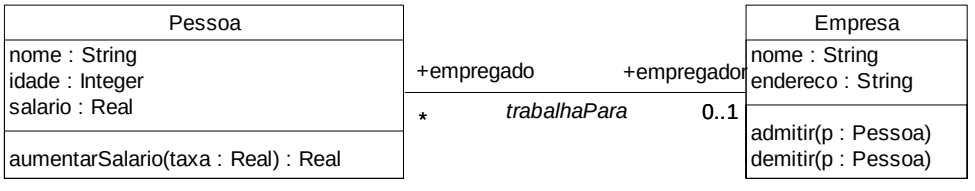


Figura 2.13 - Diagrama de Classes – Exemplo USE

A esta especificação podem ser acrescentadas restrições estruturais, bem como as pré e pós-condições das operações das classes do modelo. A Figura 2.15 ilustra a definição de uma restrição (o teto salarial dos empregados de qualquer empresa é de 15000) e das pré e pós-condições de algumas operações do modelo exemplo (operações *admitir* e *demitir* da classe *Empresa*).

```
model Exemplo
-- classes
class Pessoa
attributes
  nome : String
  idade : Integer
  salario : Real
operations
  aumentarSalario(taxa : Real) : Real
end

class Empresa
attributes
  nome : String
  endereco: String
operations
  admitir(p : Pessoa)
  demitir(p : Pessoa)
end

-- associações
association trabalharPara between
  Pessoa[*] role empregado
  Empresa[0..1] role empregador
End
```

Figura 2.14 – USE: especificação textual definida a partir do diagrama

```
constraints
context Empresa
  inv salarioMaximo: self.empregado.salario->forAll (s | s < 15000)

context Empresa::admitir(p : Pessoa)
  pre admitir_pre01: not p.oclIsUndefined()
  pre  admitir_pre02 : empregado->excludes(p)
  post admitir_post01: empregado->includes(p)

context Empresa::demitir(p : Pessoa)
  pre  demitir_pre01:  empregado->includes(p)
  post demitir_post01: empregado->excludes(p)
```

Figura 2.15 – USE: especificação de pré e pós-condições

A partir da especificação, o usuário pode modificar o estado do sistema através de comandos executados em um ambiente interativo. A Figura 2.16 ilustra o exemplo de criação de dois objetos e a posterior inicialização dos valores de seus atributos.

```
use> !create ibm : Empresa
use> !create joe : Pessoa
use> !set joe.nome= 'Joe'
use> !set joe.idade = 23
```

Figura 2.16 – USE: modificação interativa do estado do sistema

A animação do modelo é feita no mesmo ambiente interativo, através de comandos específicos para sinalizar a entrada e a saída de uma operação. A Figura 2.17 ilustra a solicitação de entrada na operação *admitir* do objeto *ibm* através do comando *!openter*. Na entrada de uma operação, o software apresenta a avaliação de cada uma de suas pré-condições.

```
use> !openter ibm admitir(joe)

precondition `admitir_pre01' is true
precondition `admitir_pre02' is true
```

Figura 2.17 – USE: indicando a entrada em uma operação

Uma vez que todas as modificações tenham sido efetuadas, o usuário indica o fim da execução simulada da operação através do comando *opexit*. Nesse momento, o software avalia todas as pós-condições da operação e apresenta o resultado. A Figura 2.18 ilustra essa chamada.

```
use> !opexit

postcondition `admitir_post01' is true
```

Figura 2.18 – USE: indicando o término de uma operação

2.6.2 Simulação por *snapshots*

Utilizamos o termo *simulação por snapshots* (LIU, 2004) para indicar a avaliação de especificações implícitas de operações de um modelo através de cenários. A especificação implícita de uma operação corresponde à definição de suas pré e pós-condições. Adaptando esse termo para o contexto de modelos UML/OCL, a *simulação por snapshot* de um modelo UML/OCL consiste em definir um espaço de objetos correspondente ao estado do sistema no momento da invocação de uma operação, e um espaço de objetos correspondente ao estado posterior à sua execução. A avaliação das pré-condições e das pós-condições da operação, bem como das restrições gerais definidas no modelo é realizada sobre esses espaços de objetos e comparada com os

resultados esperados, uma vez que especificações implícitas não podem ser executadas diretamente.

Essa avaliação também pode ser feita com uma seqüência de operações que, em (D'SOUZA e WILLS, 1998), é chamada de *filmstrip*. Um *filmstrip* corresponde à definição de uma seqüência de espaços de objetos relacionados pela execução de operações, onde cada espaço de objetos corresponde a uma configuração dos objetos e ligações entre objetos do modelo existentes em um determinado instante do tempo.

2.6.3 Modelos UML executáveis

Uma outra linha de analise dinâmica de modelos, que pode ser encontrada em algumas ferramentas de modelagem de sistemas com UML, é voltada para a execução de modelos de projeto orientado a objetos, em particular, de sistemas embarcados/tempo real. As ferramentas que seguem essa linha baseiam a execução dos modelos em diagramas de estados associados às classes e nas ações associadas aos estados e às transições de estados desses diagramas.

Uma característica dessa abordagem é que as operações são descritas de forma imperativa, seja diretamente em uma linguagem de implementação (C++, Java, etc), como nas ferramentas *I-Logix Rhapsody* (HAREL e GERY, 1997) e *HUGO* (SCHÄFER et al., 2001), seja em uma linguagem de ação mais abstrata tais como a *BridgePoint Object Action Language* (MELLOR e BALCER, 2002) e a *iUML* (KENNEDY CARTER LTD., 2002).

Uma vez que as ações são especificadas de forma explícita, o modelo pode ser avaliado através da sua execução direta. Entretanto, nenhuma dessas abordagens oferece recursos para a avaliação de restrições ou de especificações de operações do modelo produzidas com a OCL.

2.6.4 Prototipação

Prototipação é uma técnica através da qual o sistema é parcialmente construído utilizando a linguagem final de implementação ou, mais freqüentemente, uma linguagem de alto nível que possibilite o desenvolvimento rápido de pedaços do sistema. O protótipo gerado corresponde a uma implementação parcial da especificação, sendo geralmente de caráter descartável. Seu principal propósito é ser um veículo para

auxiliar no aprendizado de um problema, na validação de uma especificação ou para explorar a viabilidade de possíveis soluções (SOMMERVILLE, 1992).

Um importante benefício advindo da técnica de prototipação é a possibilidade de reduzir as diferenças de entendimento entre usuários e desenvolvedores em função de suas diferentes formações. Com um protótipo, o usuário pode exercitar o sistema como se ele estivesse no seu ambiente e, com isso, fornecer um importante retorno sobre a correção da especificação (GOMAA, 1997; SOMMERVILLE e SAWYER, 1997). Prototipação é uma das técnicas mais efetivas na identificação de problemas relacionados a requisitos (PFLEEGER, 2001).

Uma abordagem para a integração de especificações formais executáveis com a técnica de prototipação é descrita em (FENKAM et al., 2002), e consiste na utilização do IFAD VDM Toolbox, onde uma especificação VDM++ é produzida e sua execução é integrada com um protótipo de interface com usuário integrado à ferramenta de especificação através de uma API via CORBA, o que possibilita a participação do usuário no processo de validação da especificação, sem que ele necessite conhecer as notações matemáticas utilizadas na especificação.

2.7 Semântica da UML e da OCL

Um ponto positivo em relação às representações gráficas, como a UML, por exemplo, é a facilidade de entendimento que elas proporcionam. Entretanto, embora a UML ofereça uma notação expressiva, não existe uma definição semântica formal para os seus elementos. Existem vários problemas ligados a ambigüidades, contradições e omissões na especificação das várias versões da UML.

Um ponto negativo decorrente desse fato é que diferentes pessoas podem apresentar diferentes interpretações para o mesmo modelo, ou seja, o entendimento comum de um modelo é, muitas vezes, mais aparente do que real (BREU et al., 1998). Além disso, a existência de uma semântica bem definida é um requisito importante para o desenvolvimento de ferramentas de apoio à verificação e validação de modelos.

Adotamos como contexto deste trabalho um subconjunto da UML para o qual existe uma semântica bem definida. Esse conjunto é composto pela definição de classes, associações entre classes, atributos e operações de cada classe. A definição formal da

semântica da OCL passou a existir na versão 2.0 da sua especificação, e é baseada na definição formal desse subconjunto de elementos da UML (OMG, 2003a).

Assim, estamos considerando um sub-conjunto da UML representado por um esquema estrutural contendo definições de classes e associações, complementado com definições em OCL. Esse subconjunto pode ser utilizado tanto na elaboração de modelos no nível M1, como de metamodelos (nível M2). Os demais diagramas da UML como statecharts, diagramas de seqüência, entre outros, não fazem parte do escopo deste trabalho, embora a OCL também possa ser utilizada nesses diagramas.

Mesmo existindo uma definição semântica formal para esse subconjunto da UML, bem como para os elementos da OCL, existem alguns problemas relacionados tanto à linguagem como à sua definição semântica que são relevantes para a implementação de ferramentas de apoio à verificação e validação de modelos UML/OCL. A descrição detalhada dos problemas e das soluções propostas é apresentada no capítulo 7. Em síntese, podemos destacar as seguintes questões:

- Dificuldade em definir o escopo de atuação de uma operação. Este problema é conhecido como "*frame problem*" (MCCARTHY e HAYES, 1969), e a OCL não oferece recursos que facilitem a definição dos elementos de um espaço de objetos que podem ser modificados por uma operação.
- Omissões da semântica ligada à redefinição de valor inicial de atributos, regras de derivação de atributos e de operações em uma hierarquia de classes. Em particular, a definição atual da semântica da OCL não contempla a redefinição de operações.
- Ausência de recursos que permitam a reutilização local de expressões entre as diferentes pré e pos-condições de uma operação, resultando em redundância ou maior complexidade das especificações de operações.

Além disso, no contexto da elaboração de modelos executáveis, a OCL pode ser utilizada para especificar explicitamente o resultado de operações de consulta. Entretanto, como expressões OCL não podem produzir modificações no espaço de objetos onde elas sejam avaliadas, uma alternativa para a produção de especificações executáveis de operações modificadoras consiste em empregar uma linguagem de ações. A partir da sua versão 1.5, a especificação da UML passou a definir uma semântica de

ações, a UML Action Semantics (UML AS), que fornece uma forma padronizada e independente de plataforma para a especificação de ações e atividades em um modelo.

Existe, porém, uma significativa superposição entre as construções definidas na UML AS e aquelas definidas na especificação OCL, o que torna possível definir um mapeamento entre expressões OCL e uma parte significativa das ações definidas pela UML AS. Ações de leitura, ações primitivas, invocação de operações de consulta, acesso a propriedades de objetos, acesso a ligações entre objetos, bem como operações de leitura e manipulação de coleções são alguns exemplos de ações que podem ser expressas em OCL, uma vez que nenhuma dessas ações é capaz de produzir modificações no contexto de instâncias onde elas são executadas.

De um modo geral, as linguagens concretas de ação para modelos UML definem sintaxes específicas para essas ações, ignorando as construções definidas na OCL. Se considerarmos o contexto de produção de um modelo, onde sejam especificadas tanto restrições (utilizando a OCL), como ações executáveis (utilizando uma linguagem concreta de ações), a convergência sintática das duas linguagens concretas em direção à utilização de construções já presentes na OCL possibilitaria uma integração mais natural da especificação das ações com o restante do modelo, i.e., invariantes, pré-condições e pós-condições, além da possibilidade de utilização de uma biblioteca padronizada de operações de manipulação dos diferentes tipos de coleção definidos na OCL. Uma proposta para essa convergência é apresentada no capítulo 7 desta tese.

2.8 Considerações Finais

Este capítulo apresentou os principais conceitos e trabalhos relacionados que são relevantes para o entendimento dos demais capítulos desta tese. A relevância da identificação e do registro de construções problemáticas em especificações OCL foi discutida. Os trabalhos relacionados à técnica de reestruturação tanto de código, como de modelo, foram mencionados, ressaltando-se a lacuna existente na sua aplicação em especificações OCL. Um conjunto de construções problemáticas e de reestruturações aplicáveis a modelos UML/OCL que visam preencher essa lacuna é apresentado nos capítulos 3 e 4.

Este capítulo também apresentou um resumo das principais técnicas que podem ser aplicadas para a verificação e a validação de especificações de uma forma geral, e a

modelos UML/OCL, em particular. Embora existam várias ferramentas que ofereçam recursos para a avaliação sintática e para a verificação de tipos em uma especificação OCL, o suporte automatizado para análises dinâmicas ainda é bastante limitado. Não encontramos nenhuma abordagem, aplicável em modelos UML com restrições em OCL, que empregasse as técnicas de execução de modelos em conjunto com a definição de casos de teste, de forma a possibilitar a análise automática de regressão de uma especificação OCL em cenários de evolução ou de reestruturação do modelo. O capítulo 6 apresenta a proposta de uma abordagem que visa apoiar a aplicação de técnicas de análise dinâmica a modelos UML/OCL com a possibilidade da execução automática de testes de regressão sobre um modelo. A relação entre esses recursos e a execução de reestruturações manuais, bem como a utilização dessa abordagem para automatizar algumas reestruturações, são discutidas no capítulo 6.

Alguns problemas relacionados à semântica da OCL e a possibilidade de integração da OCL com a semântica de ações definida na especificação da UML também foram brevemente apresentados. Uma proposta de integração entre a semântica de ações e a OCL é descrita no capítulo 7, onde é discutida a aplicação dessa integração tanto para a produção de modelos executáveis, como para a implementação de operações sobre os modelos propriamente ditos, como é o caso das reestruturações, por exemplo.

Capítulo 3

Reestruturações OCL-Exclusivas

3.1 Introdução

O termo reestruturação pode ser definido como uma transformação de um artefato de uma forma de representação para outra, no mesmo nível relativo de abstração, que preserva a semântica do artefato transformado (CHIKOFSKY e CROSS, 1990). Reestruturação é uma técnica que desempenha um importante papel no contexto de evolução de software, uma vez que seu principal objetivo é melhorar a estrutura de um artefato, de forma a evitar que um esforço cada vez maior seja necessário nas suas futuras evoluções (MENS e TOURWÉ, 2004).

Embora a reestruturação de programas seja bastante difundida, apenas recentemente essa técnica passou a ser empregada em artefatos mais abstratos como, por exemplo, modelos UML. Entretanto, conforme descrito no capítulo anterior, nenhuma das abordagens relacionadas à reestruturação de modelos UML abrange a qualidade das restrições, especificadas em OCL, associadas aos elementos desses modelos (CORREA e WERNER, 2004a).

Não obstante a OCL ter sido definida com o objetivo de ser uma linguagem de uso mais fácil, se comparada às linguagens formais tradicionais (WARMER e KLEPPE, 2003), especificações produzidas com a OCL podem apresentar problemas de legibilidade e manutenibilidade, em função da presença de construções inadequadas, seja nas expressões OCL, seja no modelo associado. A especificação da UML 2.0 (OMG, 2005), por exemplo, emprega a OCL na definição de diversas regras para modelos bem formados. Entretanto, a presença de várias expressões complexas, redundantes, ou até mesmo incorretas, compromete a sua qualidade.

Code smells é um termo definido em (FOWLER, 1999) que corresponde a certas construções de código que podem contribuir negativamente para o entendimento ou para a evolução de um programa. De modo análogo, definimos o termo *OCL smell*

como uma construção presente em uma especificação OCL que pode contribuir negativamente para o entendimento ou para a evolução dessa especificação.

Expressões OCL que contenham problemas sintáticos ou associados à verificação de tipos, ou ainda, problemas ligados à sua semântica, não são consideradas como *OCL smells*. Desta forma, *OCL smells* envolvem especificações OCL que, embora sintática e semanticamente corretas, contenham construções que possam dificultar o seu entendimento ou a sua manutenção.

De forma análoga aos catálogos de padrões de projeto (GAMMA et al., 1995), padrões arquiteturais (BUSCHMANN et al., 1996) e de anti-padrões (BROWN et al., 1998), cada *OCL smell* catalogado é identificado por um nome, com o objetivo de criar uma terminologia comum entre os envolvidos na elaboração e na avaliação de especificações OCL.

Uma importante atividade que precede a aplicação de uma reestruturação consiste em identificar as partes de um artefato que devam ser reestruturadas. No caso de modelos contendo restrições especificadas em OCL, nossa abordagem para a identificação dessas partes é baseada na utilização de uma relação de *OCL smells*. Portanto, os *OCL smells* identificados em um modelo correspondem ao alvo das reestruturações que, através de modificações efetuadas nas expressões OCL ou no modelo associado, procuram substituí-los por construções mais apropriadas.

Esta tese descreve um conjunto de reestruturações aplicáveis a modelos ou metamodelos contendo restrições expressas em OCL, que visam remover construções com efeitos potencialmente negativos no seu entendimento e na sua evolução (CORREA e WERNER, 2006). As reestruturações propostas foram classificadas em três categorias:

- **Reestruturações OCL-exclusivas**: produzem modificações apenas em expressões OCL, ou seja, nenhum elemento é acrescentado ou modificado no modelo, seja diretamente, seja por meio de definições efetuadas através da palavra reservada *def*. Além disso, essas reestruturações não incluem a substituição de expressões por chamadas a operações ou por acessos a atributos definidos no modelo.
- **Reestruturações envolvendo atributos ou operações auxiliares**: são modificações que envolvem a utilização ou a adição de atributos ou

operações ao modelo por meio de definições efetuadas através da palavra reservada *def*. Essas reestruturações incluem a substituição de expressões por chamadas a operações ou por acessos a atributos adicionados ao modelo através dessas definições.

- **Reestruturações do modelo**: são reestruturações que demandam mudanças no modelo onde estão definidas as classes utilizadas nas expressões OCL. Adição, exclusão ou mudança de nome de classes, atributos, associações ou operações são exemplos de reestruturações desta categoria. Além disso, esta categoria inclui a substituição de expressões por chamadas a operações ou por acessos a atributos definidos diretamente no modelo.

O restante deste capítulo apresenta o conjunto de reestruturações da categoria OCL-exclusivas, bem como o conjunto de *OCL Smells* que elas visam remover de uma especificação produzida em OCL. Os *OCL Smells* e as reestruturações são descritas de forma intercalada, com o objetivo de manter a proximidade entre o problema (*OCL Smell*) e a sua respectiva solução (Reestruturação). A descrição de cada reestruturação contém: seu nome; seu objetivo; o procedimento com os passos que devem ser seguidos para que ela seja realizada e um exemplo da sua aplicação.

3.2 OCL Smell - Cadeia de Implicações

O *OCL Smell Cadeia de Implicações* corresponde a expressões que possuam a estrutura *T implies E*, conforme a gramática apresentada na Figura 3.1. Essa gramática determina que a configuração mínima para este *OCL Smell* corresponde a uma expressão da forma *A implies B implies C*, ou seja, uma expressão contendo pelo menos duas implicações em seqüencia.

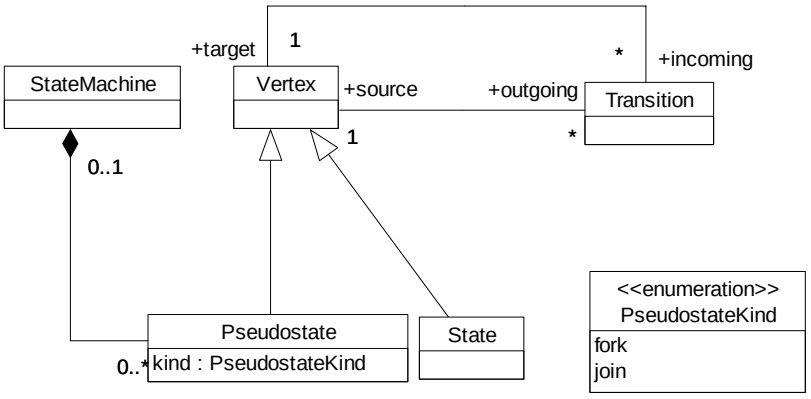
ImpliesChainSmell ::= T implies E
T ::= OCLExpression //expressão OCL do tipo Boolean
E ::= Left_par T 'implies' (T | E) Right_par
Left_par ::= '('
Right_par ::= ')'

Figura 3.1 – Gramática para o *OCL Smell* Cadeia de Implicações

A Figura 3.2 apresenta uma ocorrência deste tipo de construção, extraída da especificação da superestrutura da UML 2.0 (OMG, 2005). A restrição apresentada na

Figura 3.2 corresponde a uma versão ligeiramente modificada da restrição original, uma vez que a restrição presente na especificação original contém problemas sintáticos. Uma discussão detalhada sobre os defeitos sintáticos presentes na referida especificação pode ser encontrada em (BAUERDICK et al., 2004).

A parte superior da Figura 3.2 apresenta um fragmento do metamodelo da UML 2.0, contendo alguns elementos utilizados na definição de diagramas de transição de estados. Na parte inferior da mesma figura, a restrição associada à classe *Transition* do metamodelo foi especificada por uma expressão contendo uma cadeia de implicações. A intenção dessa restrição, definida no comentário presente nas linhas 1-4, poderia ser comunicada de forma mais direta, se o trecho correspondente às linhas 7-11 fosse substituído por uma conjunção das três expressões presentes nas linhas 7, 9 e 11.




```
 1 -- Toda transição em uma máquina de transição de estados
 2 -- (statemachine) que tiver como origem (source) um pseudo-estado
 3 -- do tipo fork, deve ter como destino (target) um elemento do tipo
 4 -- State.
 5 context Transition
 6 inv:
 7  self.source.oclIsKindOf(Pseudostate)
 8  implies
 9  (self.source.oclAsType(Pseudostate).stateMachine->notEmpty()
10  implies
11  (self.source.oclAsType(Pseudostate).kind = PseudostateKind::fork
12  implies
13  self.target.oclIsKindOf(State)))
```

Figura 3.2 – Exemplo do *OCL Smell* Cadeia de Implicações

3.3 Reestruturação – Substituir Cadeia de Implicações por uma Única Implicação

O objetivo desta reestruturação é remover o *OCL smell Cadeia de Implicações,* descrito na seção 3.2. Uma vez que uma expressão da forma b_1 *implies* (b_2 *implies* (b_3 *...* (b_{n-1} *implies* b_n))) é equivalente a uma expressão da forma (b_1 *and* b_2 *and* b_3 *...* *and* b_{n-1}) *implies* b_n (COOK et al., 2002), uma cadeia de implicações pode ser substituída por uma única implicação, onde o antecedente passa a ser uma conjunção de expressões.

Desta forma, esta reestruturação substitui uma cadeia de implicações por uma única implicação, onde o antecedente é igual à conjunção de todas as expressões, com exceção do conseqüente da última implicação da cadeia, que passa a ser o conseqüente da nova implicação.

Se a conjunção correspondente ao antecedente da nova implicação resultar em uma expressão complexa, convém definir uma propriedade ou operação auxiliar *(e.g.,* *aux* = b_1 *and* b_2 *and* b_3 *...* *and* b_{n-1}) que capture o seu propósito geral, fazendo com que a restrição final assuma a forma: *aux implies* b_n. A definição dessa propriedade ou operação auxiliar, além de simplificar a restrição, permite a sua reutilização em outras expressões, o que não seria possível com a expressão original contendo uma cadeia de implicações.

3.3.1 Procedimento

Substitua a cadeia de implicações que possua a estrutura definida na Figura 3.1, pela expressão *(T and Y) implies Z*, onde:

- *Y* corresponde à conjunção das expressões que compõem *E*, com exceção do conseqüente da última implicação definida na cadeia de implicações.
- *Z* corresponde ao conseqüente da última implicação definida na cadeia de implicações.

Dessa forma, se esta reestruturação fosse aplicada na expressão *A implies (B implies (C implies D))*, teríamos: *T = A*; *Y = B and C; Z = D*. Portanto, o resultado da reestruturação seria a expressão *(A and B and C) implies D.*

O princípio geral desta reestruturação consiste em aplicar a seguinte relação de equivalência entre fórmulas:

a => (b => c) ≡ (a and b) => c

A fórmula à esquerda dessa relação de equivalência corresponde à ocorrência mais simples do *OCL Smell Cadeia de Implicações*. Esta reestruturação aplica essa relação de equivalência de forma recursiva, substituindo uma expressão que apresente a forma à esquerda da relação pela expressão correspondente à parte da direita dessa relação.

3.3.2 Exemplo

A Figura 3.3 ilustra a aplicação desta reestruturação na restrição descrita na Figura 3.2. A restrição passou a ser expressa por uma única implicação, onde o antecedente é uma conjunção que define se a origem de uma transição é um pseudo-estado do tipo *fork*, enquanto que o conseqüente é uma expressão que indica se o destino é um elemento do tipo *State*.

```
 1 -- Toda transição em uma máquina de transição de estados
 2 -- (statemachine) que tiver como origem (source) um pseudo-estado
 3 -- do tipo fork, deve ter como destino (target) um elemento do tipo
 4 -- State.
 4 context Transition
 5 inv:
 6  (self.source.oclIsKindOf(Pseudostate) and
 7   self.source.oclAsType(Pseudostate).stateMachine->notEmpty()and
 8   self.source.oclAsType(Pseudostate).kind =
 9      PseudostateKind::fork)
10 implies
11   self.target.oclIsKindOf(State)
```

Figura 3.3 – Exemplo da reestruturação Substituir Cadeia de Implicações por uma Única Implicação

Entretanto, a complexidade da conjunção resultante revela a presença de outros *OCL smells,* conseqüencia da utilização de operações associadas à identificação de tipos (*oclIsKindOf* e *oclAsType*). Portanto, nem sempre a melhor solução para remover uma cadeia de implicações consiste na sua transformação em uma única implicação equivalente.

3.4 *OCL Smell* - Restrição Condicional Não Atômica

A OCL pode ser utilizada para especificar, de forma precisa, regras de negócio em modelos UML (ERIKSSON e PENKER, 2000). Uma regra de negócio pode ser definida como uma expressão que define ou restringe algum aspecto do negócio. Uma regra pode ser utilizada para definir aspectos estruturais do negócio, bem como para controlar ou influenciar o comportamento do negócio. Um princípio de estruturação aplicado nesse contexto consiste em definir as regras de negócio de forma atômica, de modo que cada regra represente um pensamento completo (DATE, 2000), (BAJEC e KRISPER, 2005). Esse mesmo princípio pode ser aplicado na definição de regras ou restrições associadas a modelos UML ou a meta-modelos em geral.

Algumas diretrizes devem ser seguidas para que as regras sejam definidas de forma atômica (HALLE, 2001):

- cada regra deve ter apenas um resultado;
- uma regra do tipo *SE <A> ENTÃO <B>* não deve conter conjunções na expressão correspondente ao conseqüente (*<B>*);
- uma regra do tipo *SE <A> ENTÃO <B>* não deve conter disjunções na expressão correspondente ao antecedente (*<A>*);
- duas regras não devem ser conectadas por conjunções.

Uma regra do tipo *SE <A> ENTÃO <B>*, onde *A* e *B* são expressões booleanas, pode ser definida, em OCL, por uma restrição contendo uma expressão do tipo *A implies B*.

Uma restrição condicional não atômica é definida, em OCL, por uma restrição do tipo invariante, cuja forma geral pode corresponder a uma das seguintes estruturas:

a) Uma implicação cujo antecedente é formado por disjunções de expressões*:* E_1 *or* E_2 *or ...or* E_n *implies R.*

b) Uma implicação cujo conseqüente é formado por conjunções de expressões*:* *C implies* E_1 *and* E_2 *and ... and Em.*

c) Uma expressão *if C then D else E endif*, onde *D* e *E* são expressões do tipo Boolean.

d) Uma expressão A *implies if C then D else E endif*.

Alguns exemplos de restrições condicionais não atômicas são apresentados a seguir. O primeiro exemplo, descrito na Figura 3.4, corresponde a uma restrição contendo a estrutura *SE a ENTÃO b E c,* ou seja, uma conjunção está presente no conseqüente. Nem sempre essa conjunção está especificada de forma explícita, isto é, essa configuração pode aparecer apenas se o conseqüente for expresso na forma normal conjuntiva.

```
1 -- restrição não atômica do tipo SE <A> ENTAO <B> E <C>
2 context Pedido
3 -- pedidos de clientes preferenciais têm 20% de desconto e
4 -- são entregues de forma expressa.
5 inv: self.cliente.isPreferencial() implies
6       self.desconto = 0.2 and
7       self.entrega = TipoEntrega::Expressa
```

Figura 3.4 – Exemplo de uma restrição do tipo SE a ENTÃO b E c

Caso a restrição original fosse expressa através de duas restrições atômicas (1- *SE a ENTÃO b, 2- SE a ENTÃO c)*, a mudança da restrição para, por exemplo, "apenas clientes preferenciais e que morem no Rio de Janeiro têm direito a 20% de desconto nos seus pedidos, enquanto que a entrega deve ser do tipo expressa para todos os clientes preferenciais", envolveria apenas uma das restrições atômicas.

O segundo exemplo, descrito na Figura 3.5, corresponde a uma restrição contendo a estrutura *SE a OU b ENTÃO c*, ou seja, uma disjunção está presente na expressão correspondente ao antecedente. De forma análoga ao exemplo anterior, essa configuração pode aparecer apenas se o antecedente for expresso na forma normal disjuntiva.

```
1 -- restrição não atômica do tipo SE <A> OU <B> ENTAO <C>
2 context Pedido
3 -- pedidos de clientes preferenciais ou residentes no Rio de Janeiro
4 -- são entregues de forma expressa
5 inv: (self.cliente.isPreferencial() or
6       self.cliente.isResidenteEm('Rio de Janeiro'))
7       implies
8       self.entrega = TipoEntrega::Expressa
```

Figura 3.5 – Exemplo de uma restrição do tipo SE a OU b ENTÃO c

O terceiro exemplo, descrito na Figura 3.6, corresponde a uma restrição que possui a estrutura SE c ENTÃO t_1 *E* t_2 *...E* t_n *SENÃO* e_1 *E* e_2 *... E* e_n *endif.*

```
 1  -- restrição não atômica
 2  context Pedido
 3  inv:
 4  -- pedidos de clientes preferenciais têm desconto de 20% e são
 5  -- entregues de forma expressa.
 6  -- pedidos dos demais clientes são entregues de forma normal
 7  -- e não têm desconto.
 8  if self.cliente.isPreferencial()
 9   then self.desconto = 0.2 and self.entrega = TipoEntrega::Expressa
10   else self.desconto = 0 and self.entrega = TipoEntrega::Normal
11  endif
```

Figura 3.6 – Exemplo de uma restrição contendo a estrutura if-then-else-endif

Restrições não atômicas também podem ser definidas através de estruturas condicionais mais complexas do que aquela apresentada na Figura 3.6, como, por exemplo, estruturas que contenham aninhamento de expressões condicionais em um ou mais níveis. A Figura 3.7 apresenta um exemplo genérico de uma restrição não atômica contendo aninhamento de expressões condicionais.

```
1 -- restrição não atômica com aninhamento de condições
2 -- A, B, C, E1, E2, E3 e E4 são expressões booleanas
3 context C
4 inv:
5    if A
6       then if B
7              then E1
8              else E2
9            endif
10      else if C
11              then E3
12              else E4
13           endif
14   endif
```

Figura 3.7 - Exemplo de uma restrição condicional com aninhamento

3.5 *Reestruturação* - Separar Restrições Condicionais Não-Atômicas

O objetivo desta reestruturação é modificar uma restrição contendo o *OCL smell Restrição Condicional Não Atômica*, descrito na seção 3.4, de forma que cada regra atômica seja definida como uma restrição independente.

3.5.1 Procedimento

- Para uma restrição definida por uma implicação cujo antecedente seja formado por disjunções de expressões (E_1 *or* E_2 *or ...or* E_n *implies R*), defina, para cada expressão E_i, uma nova restrição contendo a expressão E_i *implies R.*
- Para uma restrição definida por uma implicação cujo conseqüente seja formado por conjunções de expressões (*C implies* E_1 *and* E_2 *and ... and Em*), defina, para cada expressão E_i, uma nova restrição contendo a expressão *C implies* E_i.
- Para uma restrição com a estrutura *if C then D else E endif,* onde *D* e *E* são expressões do tipo Boolean, defina duas novas restrições: 1) *C implies D* e 2) *(not C) implies E.* Uma vez que a semântica da OCL define que o resultado de uma expressão *if-then-else-endif* é indefinido, caso a sua condição tenha valor indefinido, se *C* puder resultar no valor indefinido, a restrição *not C.oclIsUndefined()* deve ser adicionada.
- Para uma restrição com a estrutura *A implies if C then D else E endif*, defina duas novas restrições: 1) (*A and C) implies D* e 2) *(A and not C) implies E.* Se *C* puder resultar no valor indefinido, adicione a restrição: *A implies not C.oclIsUndefined().*
- Todas as restrições criadas devem ter o mesmo contexto e estereótipo da restrição original.
- Defina, opcionalmente, um nome para cada restrição criada.
- Para cada restrição criada que referencie variáveis definidas na restrição original, copie as expressões *let* contendo a definição dessas variáveis. Se a mesma variável for utilizada em mais de uma restrição, considere substituí-la por uma definição de atributo ou operação.

- Remova a restrição original da especificação.

É importante observar que esta reestruturação pode resultar na duplicação de expressões. A duplicação pode ocorrer no conseqüente (*R*), para as restrições com a forma descrita no item *a* do procedimento, ou no antecedente (*C*) ou (*A*), no caso das restrições com a forma descrita nos itens *b*, *c* ou *d*. Portanto, a fatoração do conseqüente ou do antecedente de regras não atômicas deve ser considerada antes da aplicação desta reestruturação.

3.5.2 Exemplos

O primeiro exemplo, ilustrado pela Figura 3.8, apresenta a decomposição da restrição descrita na Figura 3.4 em duas restrições atômicas. O conseqüente da implicação presente na restrição original era formado por uma conjunção de expressões.

```
 1 -- regras atômicas
 2 context Pedido
 3 inv Desconto_Para_Clientes_Preferenciais:
 4 -- pedidos de clientes preferenciais têm 20% de desconto
 5    self.cliente.isPreferencial() implies self.desconto = 0.2
 6 inv Tipo_de_Entrega_Para_Clientes_Preferenciais:
 7 -- pedidos de clientes preferenciais são entregues de forma
 8 -- expressa
 9    self.cliente.isPreferencial() implies
10    self.entrega = TipoEntrega::Expressa
```

Figura 3.8 – Decomposição de uma regra contendo uma conjunção de expressões no conseqüente

```
1 -- regras atômicas
2 context Pedido
3 inv Tipo_de_Entrega_Para_Clientes_Preferenciais :
4     self.cliente.isPreferencial() implies
5       self.entrega = TipoEntrega::Expressa
6
7 inv Tipo_de_Entrega_Para_ResidentesNoRJ:
8     self.cliente.isResidenteEm('Rio de Janeiro') implies
9       self.entrega = TipoEntrega::Expressa
```

Figura 3.9 – Decomposição de uma regra com uma disjunção de expressões no antecedente

O segundo exemplo, ilustrado pela Figura 3.9, apresenta uma versão reestruturada da restrição descrita no exemplo da Figura 3.5, onde o antecedente da implicação era composto por uma disjunção de expressões. Na versão reestruturada, um nome foi associado a cada restrição, visando facilitar futuras referências a essas restrições.

O terceiro exemplo ilustra a aplicação desta reestruturação à restrição descrita na Figura 3.6. A Figura 3.10 mostra que a restrição original, definida por uma estrutura do tipo *if C then* E_{a1} *and* E_{a2} *else* E_{b1} *and* E_{b2} *endif,* deu lugar a quatro regras atômicas, além de uma restrição adicional para lidar com a possibilidade de o antecedente ter o valor indefinido.

```
 1 -- regras atômicas
 2 context Pedido
 3 inv Desconto_Para_Clientes_Preferenciais:
 4 --  pedidos de clientes preferenciais têm 20% de desconto
 5     self.cliente.isPreferencial() implies self.desconto = 0.2
 6 inv Entrega_Para_Clientes_Preferenciais:
 7 --  pedidos de clientes preferenciais são entregues de forma
 8 --  expressa
 9     self.cliente.isPreferencial() implies
10        self.entrega = TipoEntrega::Expressa
11 inv Desconto_Para_Clientes_Comuns:
12 --  pedidos de clientes comuns não têm desconto
13     (not self.cliente.isPreferencial()) implies self.desconto = 0
14 inv Entrega_Para_Clientes_Comuns:
15 --  pedidos de clientes comuns são entregues de forma normal
16     (not self.cliente.isPreferencial()) implies
17        self.entrega = TipoEntrega::Normal
18 -- self.cliente.isPreferencial() não pode ter valor indefinido
19 inv: not (self.cliente.isPreferencial()).oclIsUndefined()
```

Figura 3.10 – Decomposição de uma regra if-then-else-endif

3.6 OCL Smell - Cadeia de Conjunções

Uma *Cadeia de Conjunções* corresponde a uma restrição (invariante, pré-condição ou pós-condição) composta por duas ou mais expressões conectadas pelo operador *and*. Segundo o princípio da atomicidade de regras descrito na seção 3.4, é mais adequado definir cada expressão da conjunção presente na restrição original como

uma restrição independente. Essa separação reduz a complexidade da restrição original, além de simplificar a identificação de eventuais violações das restrições durante a execução do modelo ou do código correspondente.

A restrição presente na Figura 3.11 é um exemplo deste *OCL smell*, pois contém uma restrição, definida pela conjunção de duas expressões, que poderia ter sido definida como duas restrições distintas (linhas 6 e 7, respectivamente). *Cadeia de Conjunções* é, na verdade, um caso particular do *OCL smell Restrição Condicional Não Atômica (*seção 3.4), e foi definido separadamente por ser frequentemente encontrado em especificações.

```
1  context ItemAluguel
2  -- cada item de um aluguel deve estar associado a um item
3  -- do acervo (game cd ou dvd) e a data esperada para o retorno
4  -- do item deve ser posterior à data em que o aluguel foi efetuado.
5  inv: let alugadoEm : Date = self.aluguel.alugadoEm in
6         ((self.cdJogo->notEmpty() xor self.dvd->notEmpty()) and
7          self.retornoEsperadoEm.isAfter(alugadoEm))
```

Figura 3.11 – Exemplo do *OCL Smell* Cadeia de Conjunções

3.7 Reestruturação - Separar Cadeia de Conjunções

Esta reestruturação é aplicável a restrições que apresentem o *OCL smell Cadeia de Conjunções*, que corresponde a uma expressão com a seguinte estrutura: E_1 *and* E_2 *and ... and* E_n. O objetivo desta reestruturação é definir uma restrição atômica para cada expressão que componha uma cadeia de conjunções.

3.7.1 Procedimento

- Para cada expressão E_i que componha a restrição original, defina uma nova restrição com o mesmo contexto e estereótipo da restrição original.
- Defina, opcionalmente, um nome para cada restrição criada.
- Nas restrições criadas que referenciarem variáveis definidas na restrição original, insira a definição das respectivas variáveis através de expressões *let*. Entretanto, se a mesma variável for utilizada em mais de uma das restrições criadas, considere substituí-la por uma definição de atributo ou operação.

- Remova a restrição original da especificação.

3.7.2 Exemplo

A Figura 3.12 apresenta o resultado da aplicação desta reestruturação à restrição presente na Figura 3.11. As duas expressões que formavam a conjunção presente na restrição original deram origem a duas restrições nesta versão reestruturada (linhas 2-5 e linhas 7-11 da Figura 3.12). A variável *alugadoEm* definida na versão original (linha 5 da Figura 3.11) teve que ser definida apenas na segunda restrição da versão reestruturada (linha 10 da Figura 3.12).

```
 1  context ItemAluguel
 2  inv Item_Associado_a_Item_do_Acervo:
 3  -- cada item de um aluguel deve estar associado a um item
 4  -- do acervo (game cd ou dvd)
 5      self.cdJogo->notEmpty() xor self.dvd->notEmpty()
 6
 7  inv Data_Valida_Para_RetornoEsperado:
 8  -- a data esperada para o retorno do item deve ser posterior à
 9  -- data em que o aluguel foi efetuado.
10     let alugadoEm : Date = self.aluguel.alugadoEm in
11         self.retornoEsperadoEm.isAfter(alugadoEm)
```

Figura 3.12 – Exemplo da reestruturação Separar Cadeia de Conjunções

3.8 *OCL Smell* - Cadeia de Quantificadores Universais

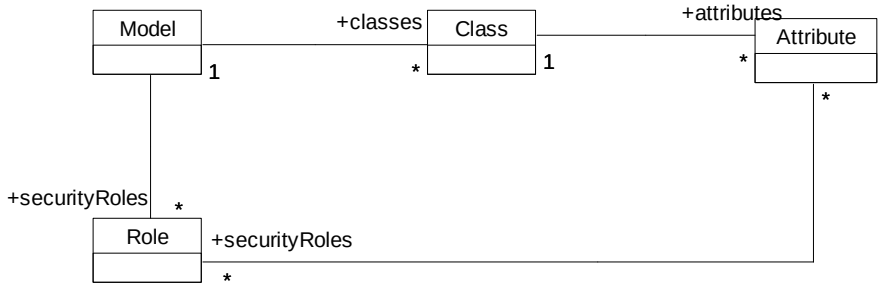
O *OCL Smell Cadeia de Quantificadores Universais* corresponde a expressões que possuam uma estrutura contendo duas ou mais chamadas aninhadas à operação *forAll.* A forma geral desse tipo de expressão é dada pelo esquema a seguir:

```
univQuantSmell ::= Collection  ForAllExp

// Collection: expressão OCL, cujo tipo é uma coleção

Collection ::= OclExpression

// ForAllExp: duas os mais chamadas à operação forAll de forma aninhada

ForAllExp ::= '->' 'forAll' '(' (Iter '|' Name '.')? Nav  (ForAllExp | InnerForAll) ')'

// Nav: expressão OCL que corresponde a uma ou mais navegações por

// associações do modelo.
```

```
    Nav ::= PropertyCallExpression
// InnerForAll: expressão forAll mais interna que não faz uso dos iteradores mais
// externos
    InnerForAll ::= '->' 'forAll' '(' (Iter)? '|' OclExpression ')'
// Iter corresponde à variável de iteração na coleção
    Iter ::= (Name | Name ':' Type)
```

A ocorrência mais simples desse tipo de expressão pode ser representada esquematicamente pela estrutura *X->forAll(x1 | x1.Y->forAll(y1 | P(y1)))*, onde *X* é uma expressão cujo tipo é uma coleção, *Y* é uma expressão de navegação pela associação entre *X* e *Y* aplicada a cada elemento *x1* de *X*, e *P(y1)* corresponde a uma expressão booleana que pode utilizar a variável *y1* na sua definição. Formas mais complexas podem ser obtidas aninhando-se outras expressões *forAll*, como, por exemplo, na expressão *A->forAll(a1 | a1.B->forAll(b1 | b1.C->forAll(c1 | P(c1))))*.

A Figura 3.13 apresenta uma parte de um modelo, adaptado de (FERNANDEZ-MEDINA e PIATTINI, 2004), onde este *OCL smell* está presente. A restrição associada à classe *Model* indica que todas as instâncias da classe *Role* (*securityRoles*) associadas a um atributo (classe *Attribute*) também devem estar associadas ao modelo indiretamente associado a esse atributo através da classe *Class*.




```
1 context Model
2 -- todos os securityRoles associados a atributos definidos no
3 -- modelo (i.e., todos os atributos de todas as classes do modelo)
4 -- devem estar definidos como securityRoles do modelo.
5 inv: self.classes->forAll(c | c.attributes->forAll(a |
6        self.securityRoles->includesAll(a.securityRoles)))
```

Figura 3.13 – Exemplo do *OCL Smell* Cadeia de Quantificadores Universais

3.9 *Reestruturação* - Substituir Cadeia de Quantificadores Universais por Navegações

O objetivo desta reestruturação consiste em simplificar expressões contendo o *OCL smell Cadeia de Quantificadores Universais,* descrito na seção 3.8.

3.9.1 Procedimento

Iniciando pela chamada mais externa à operação *forAll*:

- Certifique-se de que os iteradores associados à expressão *forAll* em questão não sejam referenciados nas expressões *forAll* mais internas.
- Substitua o fragmento da forma *X->forAll(e | e.Y->forAll(...))* por uma expressão da forma *X.Y->forAll(...),* ou seja, elimine o *forAll* mais externo, trocando-o por uma expressão de navegação que resultará em todos os elementos de *Y* associados aos elementos da coleção *X.*
- Repita os passos acima até que reste apenas uma expressão *forAll*.

3.9.2 Exemplo

A Figura 3.14 ilustra a aplicação dessa reestruturação à restrição definida na Figura 3.13. Neste exemplo, a cadeia de quantificadores universais presente na expressão *self.classes->forAll(c | c.attributes->forAll(...))* foi substituída pela expressão *self.classes.attributes->forAll(...)*, que define, de forma mais direta, a intenção da restrição expressa no comentário (linhas 2 a 4).

```
1 context Model
2 -- todos os securityRoles associados a atributos definidos no
3 -- modelo (i.e., todos os atributos de todas as classes do modelo)
4 -- devem estar definidos como securityRoles do modelo.
5 inv: self.classes.attributes->forAll(attribute |
6       self.securityRoles->includesAll(attribute.securityRoles))
```

Figura 3.14 – Exemplo da reestruturação Substituir Cadeia de Quantificadores Universais por Navegações

3.10 OCL Smell: Expressão Prolixa

Este *OCL smell* ocorre em expressões OCL definidas de forma mais extensa que o necessário. Essas expressões podem ser simplificadas sem que, para isso, seja preciso adicionar novos atributos, operações ou restrições ao modelo. Em geral, além de facilitarem o entendimento e a evolução da especificação, expressões mais simples podem ser avaliadas de forma mais eficiente.

A Figura 3.15 ilustra alguns casos freqüentes deste *OCL smell*. O último exemplo foi extraído da especificação da UML 2.0, enquanto que os demais estão representados esquematicamente: *X* corresponde a uma coleção, e *P(x)* corresponde a uma expressão booleana que pode fazer uso do elemento *x* da coleção *X* na sua definição.

```
 1) X->select(x | P(x))->size() > 0
 2) X->select(x | P(x))->size() >= 1
```

```
 3) X->select(x | P(x))->size() = 0
```

```
 4) X->select(x | P(x))->size() = 1
```

```
 5) X->forAll(x1, x2 | x1 <> x2 implies x1.p <> x2.p)
```

```
 6) X->count(object) > 0
```

```
 7) X->count(object) <> 0
```

```
 8) X->select(x | P1(x))->select(y | P2(y))
```

```
 9) X->collect(a)->collect(b)
```

```
10) X->collect(a)->sum() + X->collect(b)->sum()
```

```
11) context Namespace
    def: membersAreDistinguishable() : Boolean =
         self.member->forAll(memb |
           self.member->excluding(memb)->forAll(other |
              memb.isDistinguishableFrom(other,self)))
```

Figura 3.15 – Exemplos do *OCL Smell* Expressão Prolixa

Os sete primeiros casos correspondem a expressões que podem ser escritas de forma mais direta através do emprego de operações disponíveis nos tipos derivados de *Collection*, tais como as operações *notEmpty, isEmpty, one*, *isUnique, includes* e *excludes,* por exemplo. Os demais casos correspondem a expressões que podem ser definidas de forma mais compacta, com a utilização de um número menor de operações de coleção.

Em OCL, as restrições são sempre definidas em um contexto. Entretanto, uma restrição pode ser definida de diversas formas, uma vez que diferentes classes podem ser utilizadas como contexto para a restrição. O *OCL smell Expressão Prolixa* também pode ocorrer quando uma restrição é definida em um contexto que resulta em uma expressão mais complexa. Como diretriz geral, o melhor contexto para uma restrição é aquele que resulta na expressão mais simples (WARMER e KLEPPE, 2003).

A navegação inicial presente em uma expressão OCL é um outro aspecto que pode contribuir para a complexidade das restrições. Para ilustrar essa questão, considere os seguintes exemplos de estruturas que são frequentemente empregadas na especificação de restrições:

a) self.A->forAll(a | self.B->includesAll(a.C))
b) self.D->forAll(d | self.E->includes(d))

Ambas as estruturas são utilizadas para expressar uma relação de inclusão entre duas coleções X e Y, isto é, todos os elementos da coleção Y devem pertencer também à coleção X. Nas estruturas apresentadas no exemplo, A, B, C, D e E representam expressões genéricas que resultam em coleções de elementos. As expressões B e C resultam em coleções de elementos do mesmo tipo, o mesmo ocorrendo com as coleções resultantes das expressões D e E.

No primeiro caso, se utilizássemos *self.B* como a navegação inicial em lugar da expressão *self.A*, a restrição passaria a ser definida pela expressão *self.B->includesAll(self.A.C),* que indicaria, de forma mais direta, a intenção de que todos os elementos da coleção definida pela expressão *self.A.C* devem pertencer à coleção definida por *self.B*.

No segundo caso, se escolhermos *self.E* como a navegação inicial da expressão em lugar da expressão *self.D*, obteremos como resultado a expressão *self.E->includesAll(self.D)* que, assim como no caso anterior, indica, de forma mais direta, a intenção de que todos os elementos da coleção definida por *self.D* devem fazer parte da coleção definida por *self.E*.

3.11 Reestruturação: Simplificar Chamada de Operações

Esta reestruturação tem como objetivo remover o *OCL Smell Expressão Prolixa*, descrito na seção 3.10, nos casos onde uma expressão possa ser simplificada através do emprego de um número menor de chamadas de operação. Esta reestruturação é aplicada principalmente em expressões envolvendo as operações dos tipos coleção definidas na especificação da OCL 2.0.

3.11.1 Procedimento

- Encontre expressões que envolvam um número de operações maior do que o necessário. Em particular, investigue expressões contendo operações aplicadas a coleções. A Figura 3.16 ilustra alguns casos comuns desse tipo de expressão (coluna da esquerda) e as suas respectivas versões simplificadas (coluna da direita). Todos os exemplos estão representados esquematicamente: *X* corresponde a uma coleção, enquanto *P(x)* corresponde a uma expressão booleana que pode fazer uso da variável *x* na sua definição.
- Substitua cada expressão prolixa por sua versão simplificada.

EXPRESSÃO	VERSÃO SIMPLIFICADA
X->select(x | P(x))->size() > 0 X->select(x | P(x))->size() >= 1	X->select(x | P(x))->notEmpty()
X->select(x | P(x))->size() = 1	X->one(x | P(x))
X->select(x | P(x))->size() = 0	X->select(x | P(x))->isEmpty()
X->count(object) > 0	X->includes(object)
X->count(object) = 0	X->excludes(object)
X->select(x | P1(x))->select(y | P2(y))	X->select(x | P1(x) and P2(x))
X->collect(a)->collect(b)	X.a.b
X->collect(a)->sum() + X->collect(b)->sum()	X->collect(a + b)->sum()
X->forAll(x1, x2 | X1 <> x2 implies x1.p <> x2.p)	X->isUnique(p)
self.X->forAll(x | self.X->excluding(x)->forAll(y | P(x,y,self)))	self.X->forAll(x1, x2 | x1 <> x2 implies P(x1, x2, self))

Figura 3.16 – Exemplos de expressões prolixas e suas respectivas versões simplificadas

É importante observar que a simplificação apresentada no último caso listado na Figura 3.16 pode ser aplicada somente em expressões onde seja possível assegurar que a coleção *X* nunca terá um elemento com valor indefinido. A Figura 3.17 apresenta outros casos especiais onde, aparentemente, existe uma equivalência entre a expressão original e a sua versão simplificada.

Expressão	Versão Simplificada
X->select(x | P1(x))->size() > 0 X->select(x | P1(x))->size() >= 1 X->select(x | P1(x))->notEmpty()	X->exists(x | P1(x))
X->select(x | P1(x))->size() = 0 X->select(x | P1(x))->isEmpty()	not X->exists(x | P1(x))
X->select(x | P1(x))->forAll(y | P2(y))	X->forAll(x | P1(x) implies P2(x))
X->select(x | P1(x))->exists(y | P2(y))	X->exists(x | P1(x) and P2(x))

Figura 3.17 – Casos especiais de simplificação de expressões

As transformações apresentadas na Figura 3.17 somente podem ser aplicadas, se for possível assegurar que o predicado *P1(x)* nunca resultará no valor indefinido. Caso o predicado *P1(x)* possa resultar no valor indefinido, as expressões simplificadas passam a ter um significado diferente das suas respectivas versões originais. Para preservar a semântica original, o predicado *P1(X)* deve ser substituído pela expressão *let p: Boolean = P1(x) in not p.oclIsUndefined() and p,* o que anula, de certo modo, o efeito da simplificação.

Uma outra opção que poderia ser aplicada nos dois primeiros casos da Figura 3.17 consistiria em adicionar operações às classes coleção com essa semântica específica, de modo a simplificar o seu uso. A Figura 3.18 apresenta a definição da operação *existsDefined*, que retorna verdadeiro, caso exista algum elemento definido na coleção *source*, para o qual o valor de *exp* seja verdadeiro. Caso contrário, a operação retorna o valor falso.

Esses casos mostram que a combinação de operações de seleção com quantificadores universais ou existenciais deve ser empregada com muito cuidado, para que a intenção do especificador seja corretamente capturada pela expressão.

```
-- operação existsDefined: existe algum elemento para o qual a
-- expressão exp tenha um valor definido e igual a verdadeiro
source->existsDefined(iter | exp) : Boolean =
      source->iterate(iter; result : Boolean = true |
           result or (not exp.oclIsUndefined() and exp))
```

Figura 3.18 – Definição da operação *existsDefined*

3.11.2 Exemplos

O primeiro exemplo, ilustrado pela Figura 3.19, corresponde à reestruturação de uma expressão que define um cálculo a partir de informações coletadas a partir dos aluguéis de um cliente de uma locadora de filmes. Uma vez que essa expressão utiliza elementos coletados da mesma origem (*pCliente.alugueis)*, ela pode ser simplificada de forma a utilizar apenas uma navegação e uma chamada à operação *collect*.

```
Versão Original:
 pCliente.alugueis.itens.taxa->sum()-
 pCliente.alugueis.desconto->sum() -
 pCliente.alugueis.valorPago->sum() < 50.0
-----------------------------------------------------------------
Versão Reestruturada:
pCliente.alugueis->
  collect(itens.taxa->sum() – desconto – valorPago)->sum() < 50.0
```

Figura 3.19 – Exemplo da reestruturação Simplificar Chamadas a Operações

O próximo exemplo, ilustrado na Figura 3.20, foi extraído da especificação da infra-estrutura da UML 2.0, e corresponde à definição da operação *membersAreDistinguishable* na classe *Namespace.* A expressão original (parte superior da figura) foi simplificada através da utilização de dois iteradores (*m1* e *m2)* e de apenas um quantificador universal, resultando na versão reestruturada descrita na parte inferior da figura. Com a utilização de dois iteradores, a versão reestruturada da expressão passa a ser avaliada levando-se em consideração o produto cartesiano da coleção *self.member* com ela própria, obtendo-se o mesmo efeito da expressão *self.member->excluding(memb)->forAll(other...* , presente na expressão original.

Versão Original:

```
1 context Namespace
2 def: membersAreDistinguishable() : Boolean =
3  self.member->forAll(memb |
4    self.member->excluding(memb)->
5      forAll(other | memb.isDistinguishableFrom(other,self)))
```

Versão Reestruturada:

```
1 context Namespace
2 def: membersAreDistinguishable() :  Boolean =
3  self.member->forAll(m1, m2 |
4      m1 <> m2 implies m1.isDistinguishableFrom(m2, self))
```

Figura 3.20 – Exemplo da reestruturação Simplificar Chamada a Operações aplicada à especificação da infra-estrutura da UML 2.0.

3.12 Reestruturação: Mudar o Contexto

Esta reestruturação é motivada pela presença do *OCL Smell Expressão Prolixa*, descrito na seção 3.10, e tem como objetivo redefinir uma restrição de forma mais simples, através da utilização de um contexto diferente daquele em que a restrição foi originalmente definida.

3.12.1 Procedimento

Como um princípio geral, o contexto mais apropriado é aquele que resulta numa expressão mais simples. Portanto, o procedimento geral desta reestruturação consiste em especificar a mesma restrição utilizando diferentes contextos, até que o contexto mais apropriado seja encontrado. Geralmente, se existirem várias navegações do tipo *self.A* na restrição original, a classe associada ao papel *A* passa a ser uma candidata natural para o novo contexto da restrição.

3.12.2 Exemplo

A Figura 3.21 apresenta o resultado da aplicação desta reestruturação à restrição descrita na Figura 3.2, reproduzida novamente na parte superior da Figura 3.21. Neste exemplo, a restrição original foi redefinida no contexto da classe *Pseudostate*. Uma vez que o antecedente da expressão original referenciava apenas propriedades definidas nessa classe, a mudança de contexto resultou em uma expressão mais simples,

eliminando-se as operações de conversões de tipo (operação *oclAsType)* presentes na versão original.

Versão Original:

```
 1 context Transition
 2 inv:
 3  self.source.oclIsKindOf(Pseudostate)
 4  implies
 5  (self.source.oclAsType(Pseudostate).stateMachine->notEmpty()
 6  implies
 7  (self.source.oclAsType(Pseudostate).kind = PseudostateKind::fork
 8  implies
 9  self.target.oclIsKindOf(State)))
```

Versão Reestruturada:

```
 1 -- Toda transição em uma máquina de transição de estados
 2 -- (statemachine) que tiver como origem (source) um pseudo-estado
 3 -- do tipo fork, deve ter como destino (target) um elemento do tipo
 4 -- State.
 5 context Pseudostate
 6 inv:
 7     self.stateMachine->notEmpty() and
 8     self.kind = PseudostateKind::fork
 9 implies
10     self.outgoing.target->forAll(t | t.oclIsKindOf(State))
```

Figura 3.21 – Exemplo da reestruturação Mudar o Contexto

3.13 Reestruturação: Mudar a Navegação Inicial

Esta reestruturação é motivada pela presença do *OCL Smell Expressão Prolixa*, descrito na seção 3.10, e tem como objetivo simplificar uma expressão através da escolha de outra expressão de navegação inicial para uma restrição.

3.13.1 Procedimento

O procedimento geral desta reestruturação consiste em descrever a mesma restrição utilizando diferentes expressões iniciais de navegação, até que se obtenha a expressão mais simples.

Os casos mais comuns de aplicação desta reestruturação, descritos na Figura 3.22, correspondem ao emprego das operações *includes*, *includesAll*, *excludes*, *excludesAll* em conjunto com quantificadores universais.

Versão Original	Versão Reestruturada
self.A->forAll(a | self.B->includesAll(a.C))	self.B->includesAll(self.A.C)
self.A->forAll(a | self.B->excludesAll(a.C))	self.B->excludesAll(self.A.C)
self.D->forAll(d | self.E->includes(d))	self.E->includesAll(self.D)
self.D->forAll(d | self.E->excludes(d))	self.E->excludesAll(self.D)

Figura 3.22 – Casos comuns de mudança da navegação inicial em uma restrição

3.13.2 Exemplo

A Figura 3.23 apresenta a versão reestruturada da restrição definida na Figura 3.14, obtida através da mudança da navegação inicial de *self.classes* para *self.securityRoles*. A versão original da restrição possui a forma *self.A->forAll(a | self.B->includesAll(a.C)),* listada na Figura 3.22.

Versão Original:

```
1 context Model
2 -- todos os securityRoles associados a atributos definidos no
3 -- modelo (i.e., todos os atributos de todas as classes do modelo)
4 -- devem estar definidos como securityRoles do modelo.
5 inv: self.classes.attributes->forAll(attribute |
6       self.securityRoles->includesAll(attribute.securityRoles))
```

Versão Reestruturada:

```
1 context Model
2 -- todos os securityRoles associados a atributos definidos no
3 -- modelo (i.e., todos os atributos de todas as classes do modelo)
4 -- devem estar definidos como securityRoles do modelo.
5 inv: self.securityRoles->
6      includesAll(self.classes.attributes.securityRoles)
```

Figura 3.23 – Exemplo da reestruturação Mudar a Navegação Inicial

3.14 OCL Smell: Literal Mágico

Literal Mágico é uma construção que corresponde à utilização de uma expressão literal em uma ou mais expressões OCL de uma especificação. Na expressão apresentada na Figura 3.24, o literal 50.0 (linha 3) corresponde a uma ocorrência deste *OCL smell*. Esse literal corresponde ao valor máximo de débitos pendentes que um cliente pode ter com a locadora de filmes.

```
1 pCliente.alugueis.itens.taxa->sum()-
2 pCliente.alugueis.desconto->sum() -
3 pCliente.alugueis.valorPago->sum() < 50.0
```

Figura 3.24 – Exemplo do *OCL Smell* Literal Mágico

Literais mágicos podem tornar a especificação menos legível, além de dificultar a sua manutenção, especialmente quando o mesmo literal é utilizado em diferentes partes da especificação.

3.15 Reestruturação: Adicionar Definição de Variável

Esta reestruturação tem como objetivo adicionar definições de variáveis, de modo a facilitar a compreensão de uma restrição composta por expressões complexas. Esta reestruturação pode ser aplicada, por exemplo, para remover o *OCL smell Literal Mágico*, descrito na seção 3.14. Normalmente, esta reestruturação é utilizada em conjunto com a reestruturação *Substituir Expressão por Variável,* descrita na seção 3.16.

3.15.1 Procedimento

- Selecione a expressão a partir da qual a variável será definida (expressão origem).
- Defina o nome e o tipo da variável, de modo que o tipo da expressão origem esteja em conformidade com o tipo da variável criada.
- Defina a expressão de inicialização da variável como sendo igual à expressão origem.
- Insira a declaração da variável após a declaração de outras variáveis que sejam referenciadas pela expressão origem. A variável inserida corresponderá a uma

expressão *let,* sendo que a expressão origem fará parte, direta ou indiretamente, da expressão *in* associada a essa expressão *let.*

3.15.2 Exemplo

O exemplo da Figura 3.25 apresenta o resultado dessa reestruturação aplicada sobre a expressão presente na Figura 3.24, que resultou na definição da variável VALOR_MAXIMO_DEBITOS.

```
1 let VALOR_MAXIMO_DEBITOS = 50.0 in
2     pCliente.alugueis.itens.taxa->sum()-
3     pCliente.alugueis.desconto->sum() -
4     pCliente.alugueis.valorPago->sum() < 50.0
```

Figura 3.25 – Exemplo da Reestruturação Adicionar Definição de Variável

3.16 Reestruturação: Substituir Expressão por Variável

Esta reestruturação é uma das formas mais simples de simplificar expressões OCL ou de remover cópias de expressões ou literais mágicos. Normalmente, ela é aplicada após a reestruturação *Adicionar Definição de Variável,* descrita na seção 3.15.

Procedimento

- Defina a expressão a ser substituída (expressão origem) e a variável a ser utilizada (variável alvo). A expressão origem deve ter visibilidade da variável alvo, ou seja, a expressão origem deve ser definida em uma expressão *E* que contenha a definição da variável alvo.
- Certifique-se de que a expressão origem é igual à expressão de inicialização presente na declaração da variável alvo.
- Substitua a expressão origem por uma referência à variável alvo.

Exemplo

O exemplo ilustrado pela Figura 3.26 apresenta o resultado da aplicação dessa reestruturação sobre a expressão da Figura 3.25, onde o literal 50.0 presente no corpo da expressão (linha 4 da Figura 3.25) foi substituído pela referência à variável VALOR_MAXIMO_DEBITOS.

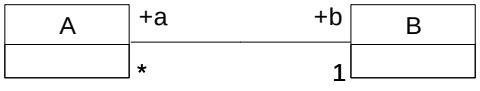
```
1 let VALOR_MAXIMO_DEBITOS = 50.0 in
2     pCliente.alugueis.itens.taxa->sum()-
3     pCliente.alugueis.desconto->sum() -
4     pCliente.alugueis.valorPago->sum() < VALOR_MAXIMO_DEBITOS
```

Figura 3.26 – Exemplo da Reestruturação Substituir Expressão por Variável

3.17 Redundância

Redundância ocorre quando uma especificação contém expressões supérfluas, isto é, expressões que podem ser removidas sem alterar a semântica da especificação. De um modo geral, expressões supérfluas introduzem múltiplos pontos de manutenção ou aumentam desnecessariamente a complexidade de uma especificação, conforme ilustrado pelos exemplos a seguir.

A Figura 3.27 apresenta um exemplo simples de redundância, onde uma restrição expressa uma regra já definida pela multiplicidade de uma associação do modelo. Nesse exemplo, a restrição associada ao classificador *A* (linhas 1-2) pode ser removida, uma vez que a multiplicidade do classificador *B* na associação *A-B* já estabelece que todo objeto *A* deve estar associado a um e apenas um objeto *B*.

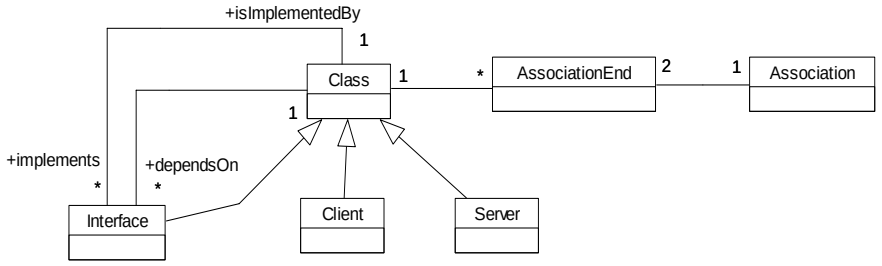



```
1 context A
2 inv: self.b->size() = 1
```

Figura 3.27 – Exemplo de Redundância: restrição x multiplicidade da associação no modelo

Um exemplo mais complexo de redundância é apresentado na Figura 3.28. Este exemplo corresponde a uma parte do metamodelo publicado em (CARIOU et al., 2004), que se destina à modelagem de aplicações cliente-servidor.

Neste exemplo, a expressão presente nas linhas 9 e 10, i.e., *servers->forAll(s | s.implements->includes(i) implies s.hasClassRefWith(self)),* pode ser suprimida da restrição, uma vez que a sua avaliação sempre resulta no valor verdadeiro, como pode ser concluído das definições da variável *servers* (linhas 2-4), da variável *interfaces* (linha 5) e da operação *hasClassRefWith* (linhas 13-15).




```
 1  context Client inv:
 2  let servers =
 3    self.associationEnd.association.associationEnd.class->
 4       select(c |  c.oclIsTypeOf(Server)) in
 5  let interfaces = self.dependsOn in
 6    servers->notEmpty() and
 7    interfaces->notEmpty() and
 8    interfaces->forAll (i | servers.implements->includes(i) and
 9          servers->forAll(s | s.implements->includes(i)
10                 implies  s.hasClassRefWith(self)))
11
12 context Class
13 def: hasClassRefWith(c1 : Class) : Boolean =
14   self.associationEnd.association.associationEnd.class->
15     exists(c | c = c1)
```

Figura 3.28 – Exemplo complexo de Redundância

A expressão *servers->forAll(s | s.implements->includes(i) implies s.hasClassRefWith(self))* somente não será avaliada como verdadeira, caso exista algum elemento *s* da coleção *servers* tal que a expressão *s.hasClassRefWith(self)* não resulte no valor verdadeiro, quando a expressão *s.implements->includes(i)* não resultar no valor falso. Como a definição da operação *hasClassRefWith* (linhas 12-15) corresponde a uma navegação no sentido inverso da navegação definida pela variável *servers* (linha 2-4), a expressão *s.hasClassRefWith(self)* será sempre avaliada como verdadeira e, portanto, a operação *forAll* em questão (linhas 9-10) terá sempre o valor verdadeiro. Dessa forma, a expressão original (linhas 8-10) poderia ser simplificada para a seguinte expressão: *interfaces->forAll (i | servers.implements->includes(i)).*

Ao contrário de vários outros *OCL smells* descritos nesta tese, *Redundância* não apresenta uma forma geral, ou seja, expressões redundantes podem se materializar de diferentes formas, conforme ilustrado pelos exemplos desta seção.

3.18 Reestruturação: Remover Redundância

Esta reestruturação tem como objetivo remover expressões ou restrições supérfluas de uma especificação, ou seja, expressões ou restrições que possam ser removidas da especificação sem alterar a sua semântica.

Procedimento

- Certifique-se de que a expressão (ou restrição) alvo pode ser removida sem alterar a semântica da especificação.
- Remova a expressão ou restrição redundante da especificação.

Exemplo

A Figura 3.29 apresenta uma versão reestruturada da restrição definida na Figura 3.28, obtida após a remoção tanto da expressão redundante *servers->forAll(s | s.implements->includes(i) implies s.hasClassRefWith(self))* (Figura 3.28 - linhas 9 e 10), como da definição da operação *hasClassRefWith* (Figura 3.28 - linhas 12 a 15). A definição desta operação foi suprimida, pois ela era utilizada apenas na expressão que foi removida.

```
1  context Client inv:
2  let servers =
3    self.associationEnd.association.associationEnd.class->
4       select(c |  c.oclIsTypeOf(Server)) in
5  let interfaces = self.dependsOn in
6    servers->notEmpty() and
7    interfaces->notEmpty() and
8    interfaces->forAll (i | servers.implements->includes(i))
```

Figura 3.29 – Exemplo da Reestruturação Remover Redundância

3.19 Considerações Finais

Este capítulo apresentou um conjunto de construções que podem contribuir de forma negativa para o entendimento e a evolução de modelos contendo especificações OCL. Essas construções deficientes foram denominadas *OCL Smells.*

Diversas reestruturações podem ser aplicadas no sentido de transformar esses *OCL smells* em construções mais adequadas. As reestruturações descritas neste capítulo correspondem àquelas que envolvem apenas a reescrita das expressões OCL, ou seja,

não é necessário acrescentar ou modificar o modelo associado, seja diretamente, seja por meio de definições efetuadas através da palavra reservada *def*.

Existem, porém, outras construções deficientes (*OCL Smells*), cuja remoção é realizada por reestruturações que envolvem a modificação, direta ou indireta, do modelo associado. Essas reestruturações podem ser realizadas através da utilização de atributos ou operações auxiliares, definidos através da construção *def* da OCL, ou então, pela adição, modificação ou eliminação de classes, atributos, operações ou associações no modelo associado às expressões OCL. O próximo capítulo descreve esses *OCL Smells* e as reestruturações que podem ser aplicadas para removê-los.

Capítulo 4

Reestruturações Envolvendo o Modelo

4.1 Introdução

Este capítulo apresenta reestruturações que envolvem atributos ou operações auxiliares, definidos através da construção *def* da OCL, reestruturações que envolvem mudanças no modelo associado, bem como o conjunto de *OCL Smells* que essas reestruturações visam remover de uma especificação produzida em OCL.

As seções 4.2, 4.3 e 4.4 apresentam os *OCL Smells* denominados *Longa Jornada*, *Exposição Indevida* e *Duplicação* que, embora se manifestem através de construções distintas, podem removidos através da aplicação das mesmas reestruturações. Essas reestruturações, descritas na seção 4.5, consistem em encapsular partes de uma expressão grande ou complexa através da definição de atributos ou operações auxiliares, utilizando a construção *def* da OCL.

As seções 4.6 e 4.7 apresentam dois *OCL Smells* relacionados à utilização de operações de identificação de tipos que, em OCL, correspondem às operações *oclIsKindOf*, *oclIsTypeOf* e *oclAsType*. Normalmente, esses *OCL Smells* podem ser removidos através da reestruturação *Introduzir Polimorfismo*, descrita na seção 4.8.

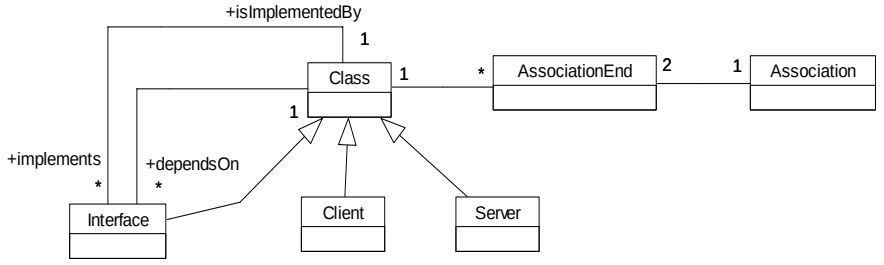
Existem situações onde a remoção de *OCL smells* demanda modificações envolvendo não apenas as expressões OCL, mas também a definição das classes do modelo associado. A seção 4.9 apresenta um conjunto de reestruturações que podem ser utilizadas nessas situações. Finalmente, a seção 4.10 apresenta as considerações finais sobre este capítulo.

4.2 *OCL Smell* - Longa Jornada

O *OCL Smell Longa Jornada* corresponde a expressões OCL que são compostas de navegações por várias associações entre diferentes classes do modelo. Quanto mais longa for uma expressão de navegação, mais ela se torna sensível a mudanças nas associações percorridas. Esse *OCL smell* é similar à violação da heurística conhecida

como Lei de Demeter (LIEBERHERR e HOLLAND, 1989).

A expressão *self.associationEnd.association.associationEnd.class* presente na linha 3 do exemplo da Figura 4.1 é um exemplo de *Longa Jornada,* pois corresponde a um encadeamento de quatro navegações por associações do modelo. Este *OCL smell* pode ser removido através da transformação de partes da navegação em propriedades ou operações das classes envolvidas.




```
1 context Class
2 def: hasClassRefWith(c1 : Class) : Boolean =
3   self.associationEnd.association.associationEnd.class->
4   exists(c | c = c1)
```

Figura 4.1 – Exemplo do *OCL Smell* Longa Jornada

4.3 OCL Smell - Exposição Indevida

Exposição Indevida ocorre quando detalhes tais como regras de inferência ou de cálculo, por exemplo, são especificados de forma explícita nas pré-condições ou nas pós-condições de operações como, por exemplo, uma *joint action* (D'SOUZA e WILLS, 1998).

De uma forma geral, este *OCL smell* se manifesta através de restrições formadas por expressões longas e complexas. De forma análoga ao que ocorre no código de programas, quanto mais extensa for uma restrição, maior a dificuldade para o seu completo entendimento. A versão 2.0 da OCL possibilita a simplificação de uma restrição complexa através da utilização de definições de propriedades e operações auxiliares nas classes envolvidas na restrição original.

Um exemplo deste *OCL smell* é apresentado a seguir. A Figura 4.2 corresponde a um fragmento do modelo de classes de um sistema de locadora de filmes e jogos.

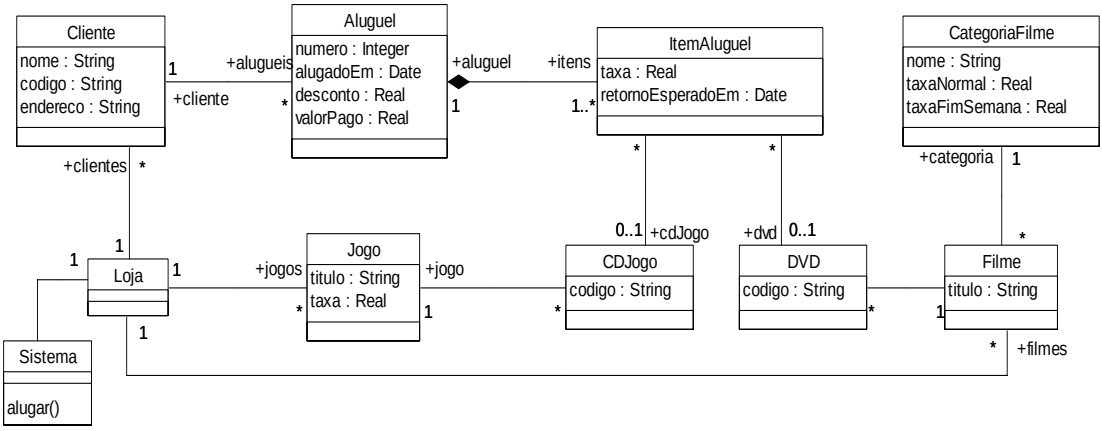


Figura 4.2 – Fragmento do modelo de classes de um sistema de locadora de filmes

A Figura 4.3 contém a especificação das pré-condições da operação *alugar* da classe Sistema, referente ao registro do aluguel de um conjunto de dvds de filmes e cds de jogos.

```
 1 context Sistema::alugar(pCliente: Cliente,
 2    pDvds : Set(DVD), pJogos: Set(CDJogo), pValorPago : Real,
 3    pDataAluguel : Date)
 4 pre:
 5 (not pCliente.oclInState(Cancelado)) and
 6 ((pCliente.alugueis.itens.taxa->sum()-
 7      pCliente.alugueis.desconto->sum()) -
 8      pCliente.alugueis.valorPago->sum() < 50.0) and
 9 (pCliente.oclInState(Normal) implies
10    pCliente.alugueis.itens->select(item |
11    item.oclInState(Pendente))->size() +
12    pDvds->size() + pJogos->size() <= 5)  and
13 (pCliente.oclInState(Premium) implies
14    pCliente.alugueis.itens->select(item |
15    item.oclInState(Pendente))->size() +
16    pDvds->size() + pJogos->size() <= 10)
```

Figura 4.3 – Exemplo do *OCL Smell* Exposição Indevida

A decisão sobre a possibilidade de um cliente alugar um conjunto de itens (dvds ou cds) envolve várias regras, que estão diretamente especificadas como pré-condições da operação *alugar,* resultando em uma especificação maior, menos legível e mais complexa dessa operação.

De forma similar ao *OCL Smell Longa Jornada* (seção 4.2), o *OCL smell Exposição Indevida* pode ser removido através da transformação de partes da expressão original em propriedades ou em operações das classes envolvidas.

4.4 *OCL Smell* - Duplicação

Duplicação é um fenômeno que pode ocorrer em diversos artefatos de software, sendo a causa de várias dificuldades durante a evolução de um software. Em OCL, o *OCL smell Duplicação* corresponde à presença de cópias da mesma expressão OCL em diferentes partes de uma especificação.

Este *OCL smell* também pode se manifestar quando duas expressões possuem várias sub-expressões iguais e uma ou mais sub-expressões ligeiramente diferentes, normalmente resultantes do processo de cópia de um conjunto de expressões, onde apenas pequenas partes da cópia são modificadas. Em geral, esse caso é um indicador de uma possível falta de generalização no modelo.

Um exemplo dessa última situação pode ser observado na Figura 4.4, que apresenta a especificação de algumas pós-condições da operação *alugar,* referente ao modelo do sistema de locadora de filmes e jogos apresentado na Figura 4.2. Os trechos correspondentes às linhas 10-21 e 22-28 definem que, para cada dvd ou cd emprestado, um novo item de aluguel (*ItemAluguel*) deve ser criado, além de especificar os valores esperados para as diversas propriedades desse novo item. Esses trechos possuem fundamentalmente a mesma estrutura, diferindo apenas em alguns detalhes dependentes do tipo de produto emprestado (dvd ou cd), o que indica que uma generalização das classes *DVD* e *CDJogo* deveria ser acrescentada ao modelo.

```
 1 context Sistema::alugar(pCliente : Cliente,
 2    pDvds : Set(DVD), pJogos : Set(CdJogo), pValorPago : Real,
 3    pDataAluguel : Date)
 4 post:
 5 pCliente.alugueis->one(novoAluguel | novoAluguel.oclIsNew()) and
 6 pCliente.alugueis->exists(novoAluguel |
 7   novoAluguel.oclIsNew() and
 8   novoAluguel.valorPago = pValorPago and
 9   novoAluguel.alugadoEm = pDataAluguel and
10   pDvds->forAll(dvd |
11      novoAluguel.itens->exists(item |
12         item.oclIsNew() and
13         item.dvd = dvd and
14         item.taxa =
15            if novoAluguel.alugadoEm.isDowBetween(
16                  DayOfWeek::Monday,DayOfWeek::Thursday)
17               then dvd.filme.categoria.taxaNormal
18               else dvd.filme.categoria.taxaFimSemana
19            endif and
20         item.oclInState(Pendente)) and
21      dvd.oclInState(Alugado)) and
22   pJogos->forAll(cd |
23       novoAluguel.itens->exists(item |
24         item.oclIsNew() and
25         item.cdJogo = cd and
26         item.taxa = cd.jogo.taxa and
27         item.oclInState(Pendente)) and
28       cd.oclInState(Alugado)) )
...
```

Figura 4.4 – Exemplo do *OCL Smell* Duplicação

Duplicação também pode se manifestar em construções específicas envolvendo expressões *if-then-else-endif*, conforme ilustrado pelo exemplo da Figura 4.5.

Neste exemplo, *A*, *B*, *C* e *E* são expressões do tipo Boolean. Como a expressão *E* está presente em uma conjunção, tanto na parte *then* como na parte *else* da expressão *if-then-else-endif*, a restrição poderia ser definida pela expressão *E and if C then A else B endif*, que, por sua vez, poderia ser decomposta nas seguintes restrições atômicas, conforme descrito na seção 3.5:

- *E*
- *C implies A*
- *not C implies B*.

```
context X
inv: if C
        then A and E
        else B and E
     endif
inv: not C.oclIsUndefined()
```

Figura 4.5 – Exemplo de Duplicação em Expressões if-then-else-endif

De forma similar ao *OCL smell Longa Jornada*, o *OCL smell Duplicação* também pode ser removido através do encapsulamento de expressões em propriedades ou operações definidas em elementos do modelo associado.

4.5 Reestruturações Envolvendo Atributos e Operações Auxiliares

A versão 2.0 da OCL (OMG, 2003a) oferece um recurso que possibilita a reutilização de expressões que sejam empregadas várias vezes ao longo de uma especificação. Esse recurso consiste na definição, em OCL, de atributos e operações que, uma vez compilados, geram atributos e operações com o estereótipo *<<OclHelper>>* associados às respectivas classes do modelo. O estereótipo *<<OclHelper>>* indica que esses elementos servem apenas como auxílio para tornar a especificação mais fácil de ser entendida e mantida, evitando a replicação de uma expressão em vários pontos.

A Figura 4.6 apresenta dois exemplos desse recurso. Nas linhas 1 e 2, um atributo auxiliar (*taxaTotal*) é definido na classe *Aluguel*, e tem o seu valor definido pela expressão *"self.itens.taxa->sum()*". Nas linhas 4 a 7, a operação *debitosAcimaDoLimite* é definida na classe *Cliente*. Toda operação com estereótipo *<<OclHelper>>* é definida como uma operação de consulta, isto é, uma operação que não pode modificar o estado do sistema. Essa operação pode ser vista como uma

informação parametrizada extraível da classe a ela associada.

```
1 context Aluguel
2 def: taxaTotal: Real = self.itens.taxa->sum()
3
4 context Cliente
5 def: debitosAcimaDoLimite(pLimite : Real) : Boolean =
6      self.alugueis->
7         collect(taxaTotal – desconto – valorPago)->sum() < pLimite
```

Figura 4.6 – Exemplos de atributos e operações com estereótipo <<OclHelper>>

Vários *OCL smells*, especialmente aqueles relacionados com expressões complexas ou duplicadas como, por exemplo, *Longa Jornada*, *Exposição Indevida* e *Duplicação*, podem ser removidos através da aplicação de reestruturações baseadas na utilização de atributos ou operações com estereótipo *<<OCL helper>>*. Esta seção apresenta as principais reestruturações desta categoria.

4.5.1 Reestruturação: Adicionar Definição de Operação

Os principais objetivos desta reestruturação são controlar a complexidade das expressões e evitar a ocorrência de duplicações ao longo de uma especificação.

4.5.1.1 Procedimento

- Identifique a expressão que dará origem à operação a ser adicionada (expressão origem).
- Defina o classificador ao qual a operação será adicionada (classificador alvo).
- Defina os parâmetros necessários para a operação. Cada parâmetro é definido por um nome e um tipo, e deve ser associado com uma expressão selecionada a partir da expressão origem. O tipo de cada expressão selecionada deve ser compatível com o tipo do respectivo parâmetro.
- Defina o nome da operação e o seu tipo de retorno. O tipo da expressão origem deve ser compatível com o tipo de retorno da operação.
- Certifique-se de que a operação a ser adicionada não contenha a mesma assinatura de outras operações definidas no classificador alvo, pois não podem existir duas operações com a mesma assinatura em um mesmo classificador.

- Defina a expressão correspondente ao objeto alvo da operação. O tipo desta expressão deve ser compatível com o classificador alvo.
- Certifique-se de que a expressão origem tenha uma árvore sintática abstrata tal que todas as expressões terminais correspondam a uma das expressões associadas com um parâmetro, à expressão referente ao objeto alvo, expressões literais ou referências a variáveis definidas na própria expressão origem através de expressões *let*.
- Defina a operação no classificador alvo. A operação deverá ter o estereótipo *<<OclHelper>>,* e o seu corpo deverá ser uma cópia da expressão origem.
- Na definição do corpo da operação, substitua cada expressão associada a um parâmetro da operação pelo nome do respectivo parâmetro.
- Na definição do corpo da operação, substitua todas as expressões iguais à expressão correspondente ao objeto alvo da operação pela palavra reservada *"self"*.

4.5.1.2 Exemplo

```
Expressão Origem:
pCliente.alugueis->
   collect(taxaTotal – desconto – valorPago)->sum() < 50.0
-----------------------------------------------------------------------
Classificador Alvo:     Cliente
Nome da Operação:       debitosAcimaDoLimite
Parâmetros:             pLimite : Real – Expressão associada: 50.0
Tipo de Retorno:        Boolean
Objeto Alvo:            pCliente
Expressões Terminais:   pCliente, 50.0
-----------------------------------------------------------------------
Operação Adicionada:
context Cliente
def: debitosAcimaDoLimite(pLimite : Real) : Boolean =
     self.alugueis->
        collect(taxaTotal – desconto – valorPago)->sum() < pLimite
```

Figura 4.7 – Exemplo da reestruturação Adicionar Definição de Operação

A Figura 4.7 apresenta um exemplo desta reestruturação. A parte superior da figura (*expressão origem*) apresenta a expressão a partir da qual a operação foi definida. A parte central descreve todas as informações relevantes para a reestruturação, e a parte inferior apresenta a operação definida na classe *Cliente*, onde as expressões *pCliente* e *50.0* da expressão original foram substituídas por *self* e *pLimite,* respectivamente.

4.5.2 Substituir Expressão por Chamada de Operação

O principal objetivo desta reestruturação é substituir expressões complexas ou expressões duplicadas por chamadas de operações. *OCL smells* como, por exemplo, *Longa Jornada* (seção 4.2)*, Exposição Indevida* (seção 4.3) e *Duplicação* (seção 4.4), podem ser removidos através da aplicação dessa reestruturação. Normalmente, esta reestruturação é aplicada após a reestruturação *Adicionar Definição de Operação* (seção 4.5.1).

4.5.2.1 Procedimento

- Identifique a expressão que deverá ser substituída por uma chamada de operação (expressão origem).
- Identifique a operação e seu classificador correspondente que será utilizada nessa substituição (operaçãoAlvo).
- Defina, a partir da expressão origem, as sub-expressões correspondentes a cada argumento da operação que será chamada (sub-exp_i).
- Defina, a partir da expressão origem, a sub-expressão correspondente ao objeto alvo da chamada da operação (objetoAlvo). O tipo dessa sub-expressão deve ser compatível com o tipo do classificador correspondente à operação alvo.
- Certifique-se de que a expressão origem é igual à expressão associada à definição da operação alvo, considerando-se a substituição dos parâmetros pelas respectivas sub-expressões, e da variável "*self*" pela expressão definida para o objeto alvo.
- Substitua a expressão origem por uma expressão da forma *objetoAlvo.operaçãoAlvo(sub-exp1, sub-exp2, ..., sub-expn)*.

4.5.2.2 Exemplo

A Figura 4.8 apresenta um exemplo de aplicação desta reestruturação. A parte superior da figura (*expressão origem*) apresenta a expressão que será reestruturada. A parte central define a operação alvo, a sub-expressão (*50.0*) que corresponderá ao parâmetro *pLimite* da operação alvo, o objeto que receberá a chamada da operação, e apresenta a definição da operação alvo após a substituição da variável *self* e dos seus parâmetros pelas expressões correspondentes. Como esta versão expandida é idêntica à expressão origem, a reestruturação pode ser aplicada, gerando a expressão apresentada na parte inferior da figura.

```
Expressão Origem:
pCliente.alugueis->
   collect(taxaTotal – desconto – valorPago)->sum() < 50.0
-----------------------------------------------------------------------
Operação Alvo:    Cliente::debitosAcimaDoLimite(pLimite:Real): Boolean
Argumentos:       50 – Parâmetro: pLimite
Objeto Alvo:      pCliente
Definição da Operação Alvo expandida:
pCliente.alugueis->
   collect(taxaTotal – desconto – valorPago)->sum() < 50.0
-----------------------------------------------------------------------
Expressão após a reestruturação:
pCliente.debitosAcimaDoLimite(50.0)
```

Figura 4.8 – Exemplo da reestruturação Substituir Expressão por Chamada de Operação

4.5.3 Adicionar Definição de Atributo / Substituir Expressão por Acesso a um Atributo

Adicionar Definição de Atributo e Substituir Expressão por Acesso a um Atributo são duas reestruturações análogas àquelas baseadas na definição de operações que foram descritas nos itens anteriores. Estas reestruturações permitem a adição de atributos com estereótipo <<*OclHelper*>> a classificadores do modelo, bem como a utilização desses atributos para simplificar expressões ou remover expressões duplicadas ao longo de uma especificação.

4.5.3.1 Procedimento

O procedimento de definição de um atributo auxiliar a partir de uma expressão é análogo àquele definido para a definição de uma operação (seção 4.5.1). Ao invés de definir uma operação com um tipo de retorno, esta reestruturação deve definir um atributo com o seu respectivo tipo. Os passos correspondentes à definição de parâmetros devem ser ignorados nesta reestruturação, uma vez que eles só valem para a definição de operações. Da mesma forma, o procedimento para substituir uma expressão por um acesso a um atributo é análogo àquele definido para a substituição de uma expressão por uma chamada de operação (seção 4.5.2).

4.5.3.2 Exemplo

A Figura 4.10 apresenta uma versão reestruturada da restrição definida na Figura 4.9. O *OCL smell Longa Jornada*, presente na definição da operação *hasClassRefWith* da versão original (linhas 2 a 4 da Figura 4.9**)**, foi substituído por uma expressão mais simples (linha 12 da Figura 4.10), que utiliza alguns atributos com estereótipo <<OclHelper>> adicionados à especificação (linhas 1 a 9 da Figura 4.10).

Versão Original:

```
1 context Class
2 def: hasClassRefWith(c1 : Class) : Boolean =
3   self.associationEnd.association.associationEnd.class->
4   exists(c | c = c1)
```

Figura 4.9 – Expressão contendo o *OCL Smell* Longa Jornada

```
 1 context Association
 2 def: participants : Bag(Class) =
 3     self.associationEnd.class
 4
 5 context Class
 6 def: associations : Bag(Association) =
 7     self.associationEnd.association
 8 def: associatedClasses : Bag(Class) =
 9    self.associations.participants
10 context Class
11 def: hasClassRefWith(c1 : Class) : Boolean =
12      self.associatedClasses->exists(c | c = c1)
```

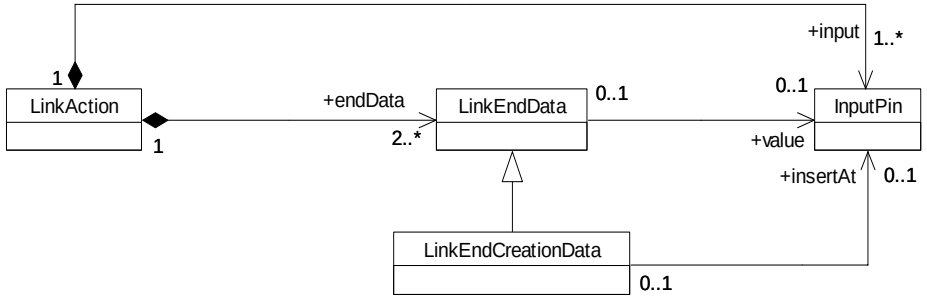
Figura 4.10 – Exemplo de reestruturações relacionadas à definição de propriedades auxiliares

4.6 *OCL Smell*: Expressões Condicionais Relacionadas a Tipos

Este *OCL smell* corresponde a expressões *if-then-else-endif* que possuam, na parte correspondente à condição da expressão, chamadas a operações que verificam se um elemento é de um determinado tipo.

A forma geral dessas expressões é dada por: *if x.oclIsKindOf(A) then <exp1> else if x.oclIsKindOf(B) then <exp2> else ... endif*, ou seja, a expressão terá o valor determinado por *<exp1>,* se a classe de *x* for *A* ou alguma especialização de *A*, *<exp2>* se a classe de *x* for *B* ou alguma especialização de *B*, e assim por diante para outras eventuais ocorrências desse tipo de condição.

Em OCL, tanto a operação *oclIsKindOf* como a operação *oclIsTypeOf* podem ser utilizadas para esse tipo de verificação. De um modo geral, a utilização desse tipo de estrutura resulta em especificações mais complexas e menos legíveis, além de indicar a possível ausência de uma estrutura de generalização no modelo.




```
 1 -- [1] os pinos de entrada de uma ação de ligação (LinkAction)
 2 -- são os mesmos pinos associados aos dados de cada extremidade de
 3 -- uma ligação mais os pinos de inserção (insertAt)
 4 context LinkAction
 5 inv: self.input =
 6 let ledpins : Set = self.endData->collect(value) in
 7    if self.oclIsKindOf(LinkEndCreationData)
 8    then ledpins->union
 9         (self.endData.oclAsType(LinkEndCreationData).insertAt)
10    else ledpins
11    endif
```

Figura 4.11 – Exemplo do *OCL Smell* Expressões Condicionais Relacionadas a Tipos

A Figura 4.11 apresenta um exemplo deste *OCL smell*, extraído da especificação da superestrutura da UML 2.0 (OMG, 2005). Além de apresentar vários erros sintáticos e de verificação de tipos, a restrição definida para a classe *LinkAction* contém uma construção *if-then-else-endif* baseada no tipo dos objetos *LinkEndData* associados a um objeto *LinkAction.*

Essa construção é utilizada na expressão presente nas linhas 6-10, cujo objetivo é definir todos os objetos *InputPin* associados aos objetos *LinkEndData* de um objeto *LinkAction*. Entretanto, como a classe *LinkEndCreationData* define um *InputPin* adicional (*insertAt*), surgem duas possibilidades para a obtenção desse conjunto:

a) para objetos da classe *LinkEndData,* o conjunto resultante corresponde à navegação de *LinkEndData* para *InputPin,* definida pelo papel *value* dessa associação (linhas 6 e 10).

b) para objetos *LinkEndCreationData,* o conjunto resultante corresponde à navegação de *LinkEndData* para *InputPin,* definida por *value*, e à navegação de *LinkEndCreationData* para *InputPin d*efinida por *insertAt*, ou seja, o conjunto resultante é igual a união dos conjuntos definidos por *value* e *insertAt (*linhas 6 a 9).

4.7 OCL Smell: Downcasting

D*owncasting* é uma construção bastante conhecida na comunidade de programação orientada a objetos, que consiste em assegurar que o resultado de uma expressão, cujo tipo seja *A,* corresponde a um elemento de um tipo *B*, onde *B* é definido como uma especialização de *A*.

Em OCL, esta construção se manifesta através da utilização de expressões contendo a estrutura *x.oclAsType(Y).z*, normalmente precedidas por uma expressão com o formato *x.oclIsKindOf(Y)*. O objetivo desse tipo de construção é tornar possível a referência a uma propriedade *z* definida em um tipo *Y,* dada uma expressão *x* de um tipo *X* do qual *Y* é uma especialização.

No exemplo apresentado na Figura 4.11, as expressões presentes nas linhas 7 e 9 correspondem a uma ocorrência deste *OCL smell*.

4.8 Reestruturação: Introduzir Polimorfismo

O objetivo desta reestruturação é evitar a utilização de expressões if-then-else-endif complexas, que façam uso de operações como *oclIsKindOf*, *oclIsTypeOf*, *oclAsType,* pois essas expressões normalmente indicam a presença de *OCL smells* como *Expressões Condicionadas a Tipos* (seção 4.6) ou *Conversão de Tipos* (seção 4.7).

4.8.1 Procedimento

- Substitua cada expressão condicional complexa por chamadas a operações através da aplicação das reestruturações *Adicionar Definição de Operação e Substituir Expressão por Chamada de Operação.*
- Defina uma operação na superclasse apropriada, e extraia sua definição da expressão original.
- Defina uma operação específica para cada uma das subclasses, e extraia sua definição da expressão original.
- Substitua a expressão original por uma chamada à operação criada na hierarquia de classes.

4.8.2 Exemplo

A Figura 4.12 apresenta a aplicação desta reestruturação à restrição definida na Figura 4.11. Uma vez que os elementos *inputPins* associados a um objeto *LinkEndData* dependem do seu tipo (*value* para instâncias de *LinkEndData* e *value* + *insertAt* para instâncias de *LinkEndCreationData)*, a operação *ledPins* foi definida em ambas as classes (linhas 1-4 e 6-9 da versão reestruturada - Figura 4.12), de modo que a restrição original pudesse ser reescrita sem utilizar as operações *oclIsKindOf* e *oclAsType* (linha 14 da versão reestruturada - Figura 4.12).

Versão Original:

```
 1 context LinkAction
 2 inv: self.input =
 3 let ledpins : Set = self.endData->collect(value) in
 4    if self.oclIsKindOf(LinkEndCreationData)
 5    then ledpins->union
 6         (self.endData.oclAsType(LinkEndCreationData).insertAt)
 7    else ledpins
 8    endif
```

Versão Reestruturada:

```
 1 context LinkEndData
 2 -- definição da operação ledPins para a superclasse LinkEndData
 3 def: ledPins() : Set(InputPin) =
 4      self.value->asSet()
 5
 6 context LinkEndCreationData
 7 -- redefinição da operação ledPins na subclasse LinkEndCreationData
 8 def: ledPins() : Set(InputPin) =
 9      self.value->union(self.insertAt->asSet())
10
11 context LinkAction
12 -- definição da restrição em LinkAction fazendo uso da definição
13 -- polimórfica da operação ledPins
14 inv: self.input = self.endData.ledPins()->asSet()
```

Figura 4.12 – Exemplo da reestruturação Introduzir Polimorfismo

4.9 Reestruturações do Modelo

Existem situações onde a remoção de *OCL smells* demanda modificações envolvendo não apenas as expressões OCL, mas também a definição das classes do modelo associado. A Figura 4.13 apresenta um exemplo de tal situação, onde a ausência de um elemento mais abstrato no modelo resulta na utilização de expressões complexas ou duplicadas na especificação de restrições.

Esse exemplo contém a especificação de uma regra associada ao modelo apresentado na Figura 4.14. Essa regra define a taxa de aluguel (*self.taxa)* de um item *(ItemAluguel)*, sendo que o valor dessa taxa depende do tipo do item alugado, neste

caso, *CDJogo* ou *DVD*. Essa dependência se reflete na estrutura condicional utilizada na definição dessa regra. A configuração presente nas linhas 3-19 da Figura 4.13 corresponde a uma ocorrência do *OCL smell Expressões Condicionais Relacionadas a Tipos*, cuja remoção, neste caso, demanda modificações no modelo.

```
 1 context ItemAluguel
 2 inv: self.taxa =
 3      if self.dvd->notEmpty()  // item corresponde a um dvd?
 4 // a taxa para empréstimo de um dvd é definida da seguinte forma:
 5 // se o empréstimo se realizar entre segunda e quinta, deve ser
 6 // cobrada a tarifa normal de empréstimo (normalFee), caso
 7 // contrário, deve ser cobrada a tarifa de final de semana
 8 //(weekendFee).
 9      then if self.aluguel.alugadoEm.dowIsBetween
10                (DayOfWeek::Monday, DayOfWeek::Thursday)
11             then self.dvd.filme.categoria.taxaNormal
12             else self.dvd.filme.categoria.taxaFimSemana
13           endif
14 // a taxa para empréstimo de um jogo é definida pela expressão:
15      else if self.aluguel.alugadoEm.month() = 1
16             then 0
17             else self.cdJogo.jogo.taxa
18           endif
19      endif
```

Figura 4.13 – Exemplo de uma expressão OCL que indica falta de generalização no modelo

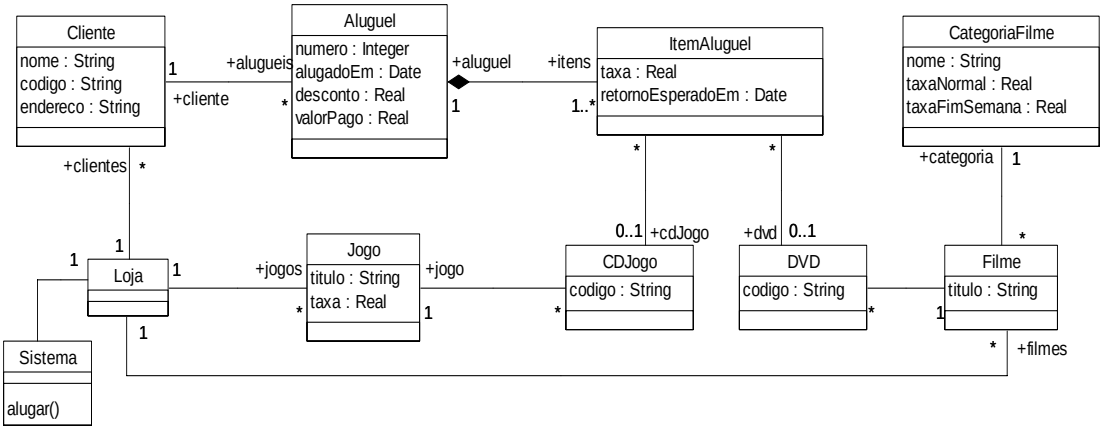


Figura 4.14 – Fragmento de um diagrama de classes de um sistema de locadora de filmes e jogos

Em (SUNYÉ et al., 2001), são descritas algumas reestruturações básicas que podem ser aplicadas a um modelo UML, em particular, à definição das classes e associações desse modelo. Essas reestruturações podem ser sintetizadas em cinco operações básicas: *adicionar*, *remover*, *mover*, *generalizar* e *especializar elementos do modelo* (classes, atributos, associações e operações).

Entretanto, essas reestruturações não levam em consideração os possíveis impactos nas restrições associadas ao modelo. Além de avaliarmos alguns desses impactos nas reestruturações descritas nesta seção, adicionamos a operação *renomear* a esse conjunto, pois ela é especialmente relevante em situações onde as restrições sejam especificadas em arquivos texto, uma vez que todas as mudanças de nomes realizadas em elementos do modelo devem ser refletidas na sintaxe concreta utilizada.

4.9.1 Adicionar atributo / associação / operação / classe

Do ponto de vista sintático, a adição de elementos a um modelo não gera interferência sintática nas restrições já existentes. Do ponto de vista semântico, entretanto, a adição de um elemento deve ser realizada após a avaliação das suas implicações em relação à necessidade de modificações no conjunto de restrições, ou ainda, à possibilidade de inconsistência com restrições já existentes. A adição de uma associação entre duas classes, por exemplo, pode levar à adição de novas restrições envolvendo essa associação, modificações nas especificações de operações que envolvam objetos das classes envolvidas na associação, dentre outros efeitos possíveis.

A adição de um atributo a uma classe *A* somente poderá ser realizada quando não existir uma propriedade com o mesmo nome na classe *A*, ou em alguma classe ancestral ou descendente de *A*. A adição de uma associação entre duas classes (*A* e *B*) somente poderá ser efetuada se não existir nenhuma propriedade definida em *A*, ou em alguma classe ancestral ou descendente de *A*, com o mesmo nome atribuído ao papel de *B* nessa associação, e não existir nenhuma propriedade definida em *B*, ou em alguma classe ancestral ou descendente de *B*, com o mesmo nome atribuído ao papel de *A* nessa associação.

Do ponto de vista sintático, uma operação poderá ser adicionada a uma classe *A* se não existir, nessa classe, nenhuma operação com a mesma assinatura. Em um programa, para adicionarmos um método a uma subclasse com a mesma assinatura

definida na sua superclasse, a definição do método da subclasse deve ser compatível com a definição presente na superclasse (ROBERTS, 1999). Em modelos UML, a mesma regra pode ser aplicada para a adição de operações. Entretanto, enquanto a equivalência estrutural entre atributos, associações e assinatura de operações pode ser verificada através de simples comparações sintáticas, a equivalência semântica entre métodos ou operações é, em geral, um problema bastante complexo (SUNYÉ et al., 2001).

A adição de uma classe pode ser efetuada sem interferência sintática nas expressões OCL associadas ao modelo. Uma classe *A* também pode ser adicionada como subclasse de uma classe *B* já existente no modelo. Uma classe *A* somente pode ser adicionada ao modelo, se não existir outra classe *B* com o mesmo nome no espaço de nomes onde a classe *A* será definida.

4.9.2 Remover atributo / associação / operação / classe

A remoção de um elemento do modelo somente poderá ser realizada quando esse elemento não for referenciado por outros elementos ou restrições. No caso de modelos com especificações OCL, isso implica nas seguintes situações relacionadas à associação entre os elementos de um modelo de classes e os elementos correspondentes às expressões OCL:

- Uma classe só poderá ser removida se nenhum outro elemento estiver associado a ela. Isso implica que a classe não pode ter nenhum atributo ou operação, participar de nenhuma associação ou ser referenciada por qualquer expressão OCL.
- Um atributo só poderá ser removido se nenhum outro elemento estiver associado a ele. No caso de expressões OCL, não pode existir nenhuma instância de *AttributeCallExp* associada a esse atributo, ou seja, não pode existir nenhuma expressão OCL referenciando esse atributo.
- Uma associação só poderá ser removida se nenhum outro elemento estiver associado a ela. No caso de expressões OCL, não pode existir nenhuma instância de *NavigationCallExp* ligada a elementos dessa associação, ou seja, não pode existir nenhuma expressão OCL que navegue por essa associação.
- Uma operação poderá ser removida se nenhum outro elemento estiver

associado a ela. No caso de expressões OCL, não pode existir nenhuma instância de *OperationCallExp* ligada a essa operação. Caso existam expressões de chamada de operação referenciando essa operação, ela somente poderá ser removida de sua classe *(C)* se existir, em algum ancestral de C, uma operação com a mesma assinatura e com a mesma semântica. Nesse caso, as ligações existentes entre as instâncias de *OperationCallExp* e a operação a ser removida serão eliminadas, e serão criadas ligações entre essas instâncias de *OperationCallExp* e a operação definida no ancestral.

4.9.3 Renomear atributo / associação / operação / classe

Renomear um elemento do modelo implica apenas na mudança do valor da sua propriedade *name* e, portanto, esta reestruturação não afeta as instâncias de elementos definidos pelo metamodelo da OCL correspondentes às expressões OCL associadas ao modelo. O impacto desta reestruturação é restrito apenas à representação textual das expressões na sua sintaxe concreta que, uma vez compiladas, dão origem a essas instâncias de elementos do metamodelo OCL.

4.9.4 Mover atributo ou operação para classe ancestral

Esta reestruturação consiste em mover, para uma classe *A,* a definição de um atributo ou operação presente em uma ou mais classes descendentes de *A*. As restrições ligadas ao espaço de nomes, descritas para a adição de elementos (seção 4.9.1), devem ser respeitadas nesta reestruturação. O atributo ou a operação a ser movida deve possuir a mesma assinatura em todas as classes envolvidas.

A movimentação de um atributo para uma classe ancestral não produz nenhuma alteração nas expressões OCL que referenciem o elemento movido, uma vez que ela apenas amplia o escopo de utilização desse elemento. Entretanto, uma limitação imposta pelo metamodelo da OCL pode impossibilitar essa operação: sejam duas classes *A1* e *A2*, subclasses de *A*, e suponha que um atributo *att* esteja definido em *A1* e *A2*. Se *A1* e *A2* definirem expressões de valor inicial distintas para o atributo *att*, esse atributo não poderá ser movido para *A,* já que a expressão de valor inicial, de acordo com o metamodelo, é associada diretamente ao atributo e, portanto, não pode ser redefinida em cada sub-classe. Essa limitação, bem como uma proposta de solução, será discutida no

capítulo 7.

A movimentação de uma operação para uma classe ancestral também não produz nenhuma alteração sintática nas expressões OCL que referenciem essa operação. Entretanto, se a movimentação envolver duas ou mais operações de classes descendentes que contenham restrições associadas (pré-condições, pós-condições ou corpo), ela somente poderá ser realizada se as restrições presentes nas operações das classes descendentes forem equivalentes e, portanto, puderem ser definidas apenas na classe ancestral.

4.9.5 Mover atributo ou operação para classes descendentes

Esta reestruturação produz o efeito oposto à reestruturação *Mover atributo ou operação para classe ancestral* (seção 4.9.4), ou seja, um elemento de uma classe *A* é definido em cada classe pertencente a um conjunto *D* de classes descendentes de *A*.

Para que a reestruturação *Mover atributo para classes descendentes* possa ser efetuada, o tipo da expressão origem, em todas as expressões do tipo *AttributeCallExp* que utilizem o referido atributo, deve ser uma classe do conjunto *D* ou descendente de uma classe de *D.*

A reestruturação *Mover operação para classes descendentes* é similar à reestruturação *Mover atributo para classes descendentes* e, de forma análoga, o tipo da expressão origem, em todas as expressões do tipo *OperationCallExp* que utilizem a referida operação, deve ser uma classe do conjunto *D* ou descendente de uma classe de *D.* Todas as restrições definidas na operação a ser movida (pré-condições, pós-condições e corpo) devem ser definidas em cada operação criada nas classes descendentes.

4.9.6 Mover atributo

Um atributo pode ser movido de uma classe *A* para uma classe *B,* desde que exista uma associação entre *A* e *B* de multiplicidade 1 tanto em *A* como em *B*, e que essa movimentação respeite as restrições ligadas à duplicação de nomes, descritas na seção 4.9.1.

As expressões OCL que referenciem esse atributo precisam ser modificadas em função da movimentação do atributo. Dessa forma, uma expressão *exp.attribA*, onde

attribA é um atributo originalmente definido na classe *A,* e que passará a fazer parte da classe *B* após a reestruturação, deve ser modificada para *exp.b.attribA*, onde *b* corresponde à navegação de *A* para *B*, considerando que a multiplicidade da associação seja 1.

4.9.7 Mover operação

De forma análoga à reestruturação *Mover Atributo,* uma operação pode ser movida de uma classe *A* para uma classe *B,* desde que exista uma associação entre *A* e *B* de multiplicidade 1 tanto em *A* como em *B*, e que essa movimentação respeite as restrições ligadas à duplicação de nomes, descritas na seção 4.9.1. Entretanto, a movimentação de uma operação envolve alguns passos adicionais se comparada com a movimentação de um atributo:

- Como a mudança de uma operação de uma classe *A* para uma classe *B* envolve uma mudança do classificador que contextualiza as restrições associadas à operação, é necessário alterar as expressões dessas restrições que façam referência à variável *self,* introduzindo uma navegação de *self* para a instância da classe *B* associada à classe *A*. Dessa forma, uma expressão *self.exp* seria substituída por uma expressão *self.b.exp* nas restrições associadas a essa operação.
- Nas expressões que referenciem essa operação, ou seja, expressões da forma *exp.op()*, uma navegação para a classe *B* deve ser adicionada. Portanto, toda expressão com a forma *exp.op()* deve ser alterada para *exp.b.op()*.

4.9.8 Adicionar Generalização

A adição de um classificador também pode ser efetuada em uma hierarquia de classes. Esta reestruturação consiste em criar uma classe vazia em uma estrutura de classes já existente, de forma que, se duas classes *A* e *B* forem especializações de *C*, ao inserirmos uma classe genérica *D*, *A* e *B* passam a ser especializações de *D*, enquanto *D* passa a ser uma especialização de *C*. Essa reestruturação não afeta as restrições associadas ao modelo.

4.9.9 Remover Generalização

Esta reestruturação realiza a operação inversa da reestruturação *Adicionar Generalização*, ou seja, remove uma superclasse *D* do modelo, e define uma relação de generalização entre as suas subclasses e a sua superclasse. Como em qualquer operação que envolva remoção de um elemento, este não pode ser referenciado por outro elemento do modelo, ou por qualquer elemento de uma expressão OCL.

4.9.10 Outras Reestruturações

As reestruturações definidas na seção 4.5 podem ser adaptadas para utilizar operações de consulta ou atributos derivados definidos diretamente no modelo de classes, ao invés de utilizar atributos ou operações auxiliares definidos por expressões *def*.

Para adicionarmos uma definição de uma operação ao modelo a partir de uma expressão, o procedimento é idêntico àquele definido para a reestruturação *Adicionar Definição de Operação* (seção 4.5.1), exceto no passo onde uma operação com estereótipo <<*OclHelper*>> é definida através de uma expressão *def*. Esse passo deve ser substituído pela adição da definição de uma operação de consulta ao modelo (atributo *isQuery* = true), e pela definição do seu corpo através da palavra reservada *body* em lugar da palavra reservada *def*.

A Figura 4.15 apresenta como a operação *debitosAcimaDoLimite* poderia ser definida, se fosse acrescentada diretamente à classe *Cliente*, ao invés de utilizar uma expressão *def*, como no exemplo apresentado na Figura 4.6.

```
1 context Cliente:: debitosAcimaDoLimite(pLimite : Real) : Boolean
2 body:
3   self.alugueis->
4      collect(taxaTotal – desconto – valorPago)->sum() < pLimite
```

Figura 4.15 – Exemplo de definição do corpo de operações de consulta adicionadas diretamente ao modelo de classes a partir de uma expressão

De forma análoga, ao invés de adicionarmos a definição de um atributo auxiliar através da palavra reservada *def* (seção 4.5.3), podemos adicionar um atributo derivado a uma classe do modelo (atributo isDerived = true), e associar uma expressão de derivação através da palavra reservada *derived*, como ilustrado pelo exemplo da Figura

4.16. Esse exemplo mostra a definição da propriedade *participants*, definida originalmente na Figura 4.10, como um atributo derivado na classe *Association*.

```
1 context Association::participants : Bag(Class)
2 derive:  self.associationEnd.class
```

Figura 4.16 – Exemplo de definição de um atributo derivado adicionado diretamente ao modelo de classes a partir de uma expressão

As reestruturações *Substituir Expressão por Chamada de Operação* (seção 4.5.2) e *Substituir Expressão por Acesso a um Atributo* (seção 4.5.3) também podem ser aplicadas com operações de consulta e atributos derivados do modelo. O procedimento de aplicação é o mesmo, exceto pela forma de acesso à definição desses elementos que será utilizada na comparação com a expressão a ser substituída.

O mesmo vale para a reestruturação *Introduzir Polimorfismo* (seção 4.8), que pode ser aplicada pela definição de operações diretamente no modelo, e pela substituição de expressões por chamadas a essas operações.

4.9.11 Exemplo de Aplicação

Esta seção apresenta um exemplo de como as várias reestruturações apresentadas neste capítulo podem ser aplicadas para remover *OCL smells* de um modelo. Esse exemplo descreve a reformulação da especificação apresentada na Figura 4.13 através da aplicação da seqüência de reestruturações descrita a seguir:

a) Efetuar a reestruturação *Adicionar Definição de Operação,* onde a operação *taxaAluguel(data : Date) : Real* é adicionada à classe *DVD*, e o seu corpo é definido a partir da expressão *if-then-else-endif* presente nas linhas 9-13 da Figura 4.13.

b) Efetuar, novamente, a reestruturação *Adicionar Definição de Operação,* desta vez, para adicionar a operação *taxaAluguel(data : Date) : Real* à classe *CDJogo*. O corpo dessa operação é definido a partir da expressão presente nas linhas 15-18 da Figura 4.13. A Figura 4.17 ilustra o resultado parcial obtido com essas duas reestruturações.

```
 1 context DVD::taxaAluguel(data : Date) : Real
 2 body:
 3    if data.dowIsBetween(DayOfWeek::Monday, DayOfWeek::Thursday)
 4       then self.filme.categoria.taxaNormal
 5       else self.filme.categoria.taxaFimSemana
 6    endif
 7 context CDJogo::taxaAluguel(data : Date) : Real
 8 body:
 9    if data.month = 1
10       then 0
11       else self.jogo.taxa
12    endif
```

Figura 4.17 – Operação taxaAluguel adicionada ao modelo da locadora de filmes.

c) Efetuar a reestruturação *Substituir Expressão por Chamada de Operação* na expressão correspondente à cláusula *then* por uma chamada à operação *taxaAluguel* de *dvd.*

d) Efetuar a reestruturação *Substituir Expressão por Chamada de Operação* na expressão correspondente à cláusula *else* por uma chamada à operação *taxaAluguel* de *gameCD.* A Figura 4.18 apresenta o resultado da aplicação dessas reestruturações.

```
 1 context ItemAluguel
 2 inv: self.taxa =
 3     if dvd->notEmpty()
 4        then self.dvd.taxaAluguel(aluguel.alugadoEm)
 5        else self.cdJogo.taxaAluguel(aluguel.alugadoEm)
 6     endif
```

Figura 4.18 – Substituição das expressões por chamadas à operação taxaAluguel

e) Efetuar a reestruturação *Adicionar Generalização,* onde um classificador genérico e abstrato (*ItemAcervo)* é definido como uma superclasse de *CDJogo* e *DVD*.

f) Efetuar a reestruturação *Mover atributo para classe ancestral,* onde o atributo *codigo* é movido das classes *CDJogo* e *DVD* para a classe *ItemAcervo.*

g) Efetuar a reestruturação *Adicionar operação*, onde a operação abstrata *taxaAluguel(data : Date) : Real* é acrescentada à classe *ItemAcervo.*

h) Efetuar a reestruturação *Adicionar associação*, onde uma associação entre as classes

ItemAluguel e *ItemAcervo* é adicionada ao modelo.

i) Efetuar a reestruturação *Introduzir Polimorfismo*, onde o *OCL Smell Expressões Condicionais Relacionadas a Tipos* é removido da restrição original. A expressão apresentada na Figura 4.18 é substituída por uma chamada polimórfica à operação *taxaAluguel*. A Figura 4.19 apresenta a expressão resultante.

```
1 context ItemAluguel
2 inv: self.taxa = self.item.taxaAluguel(aluguel.alugadoEm)
```

Figura 4.19 – Versão reestruturada da especificação da regra de cálculo de empréstimo de itens

j) Considerando que não exista mais nenhuma referência da classe *ItemAluguel* para os elementos *dvd* e *cdJogo*, aplicamos a reestruturação *Remover Associação*, onde a associação entre as classes *ItemAluguel* e *DVD* é removida, assim como a associação entre as classes *ItemAluguel* e *CDJogo*.

A Figura 4.20 apresenta o modelo resultante dessas reestruturações.

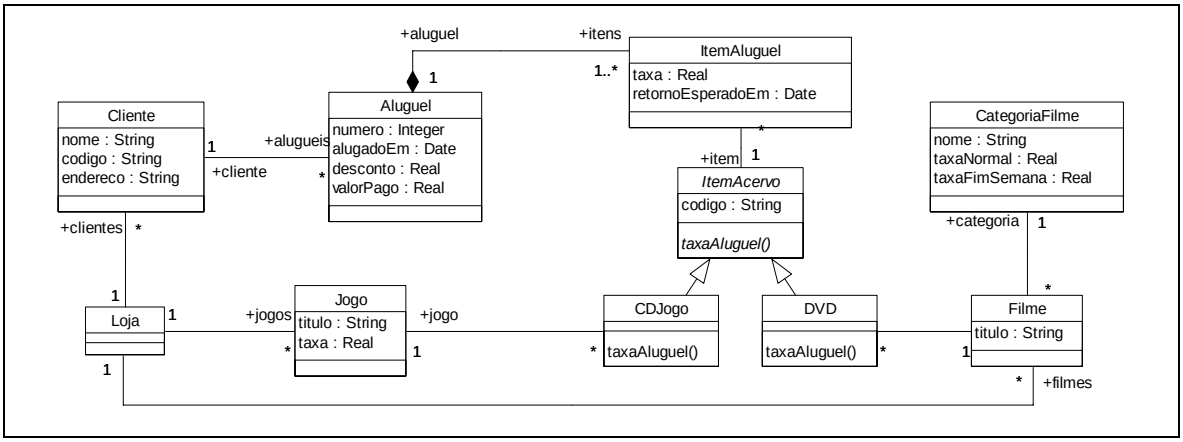


Figura 4.20 – Modelo reestruturado do sistema de locadora de vídeo

4.10 Considerações Finais

Este capítulo descreveu um outro conjunto de construções deficientes (*OCL Smells*), cuja remoção é realizada por reestruturações que envolvem a modificação, direta ou indireta, do modelo associado. Essas modificações podem ser realizadas com a definição de atributos ou operações auxiliares, através do emprego da construção *def* definida na OCL, ou com a adição, modificação ou eliminação de classes, atributos, operações ou associações no modelo associado.

Alguns *OCL smells* descritos neste capítulo estão relacionados com os *code smells* definidos em (FOWLER, 1999). O *OCL smell Duplicação*, por exemplo, é uma adaptação do *code smell Duplicated Code* para especificações declarativas em OCL. O *OCL smell Longa Jornada* é uma adaptação do *code smell Message Chain,* enquanto que o *OCL smell Expressões Condicionais Relacionadas a Tipos* é uma adaptação do *code smell Switch Statements*.

Conforme pode ser observado pelos exemplos descritos neste capítulo e no capítulo anterior, um *OCL smell* pode ser removido de várias formas, isto é, diferentes reestruturações podem ser aplicadas a certos *OCL smells*. Além disso, a remoção de um *OCL smell* pode envolver a aplicação de diversas reestruturações, como foi o caso do exemplo apresentado na seção 4.9.11.

Várias reestruturações descritas neste capítulo estão relacionadas com as reestruturações descritas em (OPDYKE, 1992), (FOWLER, 1999) e (SUNYÉ et al., 2001). As reestruturações no modelo associado (seção 4.9) são adaptações de reestruturações descritas em (OPDYKE, 1992) e (SUNYÉ et al., 2001), que passaram a considerar os possíveis impactos que a presença de restrições associadas aos elementos do modelo pode ocasionar. Uma descrição formal das condições necessárias para as reestruturações descritas na seção 4.9 pode ser encontrada em (MASSONI et al., 2005).

Embora várias das reestruturações propostas neste capítulo sejam primitivas, e.g., *Adicionar Definição de Operação* e *Substituir Expressão por Chamada de Operação*, elas podem ser combinadas de diferentes maneiras, no sentido de gerar reestruturações mais complexas. Esta separação tem como objetivos possibilitar que as reestruturações sejam definidas de forma mais simples e compacta, além de possibilitar a definição de novas reestruturações a partir dessas reestruturações primitivas, de forma similar àquela descrita em (OPDYKE, 1992).

A ocorrência de certos *OCL smells* pode indicar a necessidade de melhorias no modelo associado. Esse é o caso, por exemplo, de estruturas duplicadas ou de expressões condicionais baseadas em tipos de objetos, que normalmente são indicadores da ausência de certas propriedades, operações ou de construções mais genéricas no modelo. De certa forma, a elaboração de especificações OCL e a ocorrência desses *OCL smells* podem antecipar a descoberta desse tipo de problema no modelo.

Os *OCL smells* descritos neste capítulo e no capítulo anterior não representam uma relação exaustiva de todas as possíveis construções deficientes que podem ser encontradas em uma especificação elaborada em OCL. Eles representam as construções deficientes encontradas com maior freqüência nas especificações que foram analisadas. A mesma observação vale para as reestruturações descritas nestes dois capítulos.

Portanto, tanto o conjunto de *OCL smells* quanto o de reestruturações devem estar em constante evolução. Eles representam um primeiro passo no sentido de registrar um conhecimento importante para a comunidade envolvida na produção de especificações de modelos, de regras de boa formação de meta-modelos, ou de qualquer outra atividade que envolva a produção de expressões ou restrições elaboradas em OCL.

O capítulo a seguir apresenta o estudo experimental realizado para avaliar os efeitos da aplicação de reestruturações em expressões contendo *OCL smells* no entendimento de restrições especificadas em OCL.

Capítulo 5

Estudo Experimental

5.1 Introdução

Após a definição dos *OCL smells* e das reestruturações propostas para removê-los, um estudo experimental foi planejado e executado com o objetivo de identificar indícios de viabilidade da utilização das técnicas de reestruturação em restrições especificadas em OCL. Em particular, nosso objetivo foi obter indícios de que a aplicação de reestruturações pode trazer benefícios em relação à facilidade de entendimento de especificações produzidas com a OCL. Neste capítulo, apresentamos o plano, os resultados e as lições aprendidas deste estudo.

Em um estudo de viabilidade, os dados são coletados de acordo com algum planejamento experimental, embora o controle sobre todas as possíveis variáveis não possa ser atingido (SHULL et al., 2001). Este tipo de estudo visa oferecer informações que possam apoiar a decisão sobre um possível aprimoramento das técnicas propostas.

O restante deste capítulo está organizado em sete seções. Na seção 5.2, apresentamos a definição do estudo experimental, ressaltando seus objetivos. Na seção 5.3, mostramos o planejamento do estudo. Na seção 5.4, apresentamos a execução do estudo, explicitando as características e a organização dos seus participantes. Na seção 5.5, apresentamos os resultados da análise dos dados obtidos com o estudo. Na seção 5.6, discutimos as ameaças à validade dos resultados obtidos. A seção 5.7 apresenta algumas lições aprendidas com o estudo e, finalmente, a seção 5.8 apresenta as considerações finais deste capítulo.

5.2 Definição do Estudo

O objetivo deste estudo foi verificar se a compreensão de restrições especificadas em OCL pode ser afetada pela estrutura das expressões que a compõem. Mais especificamente, queríamos observar se a presença de *OCL smells* exerce influência na compreensão das restrições, quando comparada com suas respectivas versões obtidas a

partir da aplicação das reestruturações definidas nos capítulos 3 e 4. Este estudo foi desenvolvido sob a ótica do pesquisador, e envolveu 22 profissionais de desenvolvimento de software, todos pós-graduados na área de desenvolvimento de software, utilizando um pacote definido em laboratório contendo modelos UML e restrições especificadas em OCL. A estrutura a seguir sintetiza a definição do estudo:

> **Analisar** o efeito da aplicação de reestruturações em restrições especificadas em OCL que contenham *OCL smells*;
>
> **Com o propósito** de caracterizar a viabilidade do uso e da continuidade do desenvolvimento desses conceitos;
>
> **Referente** aos benefícios obtidos em relação ao entendimento de restrições especificadas em OCL;
>
> **Do ponto de vista** do pesquisador;
>
> **No contexto de** profissionais da indústria, pós-graduados na área de desenvolvimento de software, interpretando restrições especificadas em OCL.

5.3 Planejamento do Estudo

5.3.1 Seleção do contexto

Este estudo consistiu em uma atividade de leitura e interpretação de restrições especificadas em OCL sobre um modelo UML especialmente preparado em laboratório. Portanto, a atividade realizada pelos participantes não estava no contexto de um projeto real da indústria. Os participantes deste estudo são, em sua grande maioria, profissionais com pelo menos três anos de experiência em desenvolvimento de software na indústria.

A estrutura das expressões OCL foi o principal fator de interesse deste estudo. Desta forma, os participantes foram divididos em dois grupos, e todos responderam a um questionário contendo dois tipos de questões:

- tipo S: questões de interpretação de restrições com a presença de *OCL smells;*

- tipo R: questões de interpretação de restrições obtidas a partir de reestruturações aplicadas às expressões presentes nas questões do tipo S do questionário aplicado ao outro grupo.

5.3.2 Formulação das Hipóteses

Hipótese Nula: a hipótese nula determina que o entendimento das restrições de um modelo não é influenciado pela sua estrutura, ou seja, não há diferença na correção e no esforço despendido no entendimento de restrições contendo *OCL smells*, em comparação com as restrições obtidas através das reestruturações descritas nesta tese.

A correção do entendimento de um participante é medida pelas variáveis PS e PR, que correspondem ao número total de pontos obtidos nas questões dos tipos S e R, respectivamente. A forma de pontuação das questões é descrita na seção 5.3.4. O tempo gasto por um participante na resolução das questões é medido pelas variáveis TS e TR, que correspondem às questões do tipo S e R, respectivamente. Dessa forma, a hipótese nula é definida em função das médias dos pontos obtidos e dos tempos gastos pelos participantes na resolução de questões dos dois tipos de tratamento (S e R):

H_O: $\mu PS = \mu PR$ e $\mu TS = \mu TR$

Hipótese Alternativa: existe diferença na correção ou no esforço despendido no entendimento das restrições de um modelo em função da estrutura apresentada por essas restrições. Mais especificamente, o número de pontos obtidos pelos participantes em questões formuladas sobre expressões OCL é maior quando elas estão bem estruturadas e não apresentam *OCL smells,* e/ou o tempo gasto na resolução das questões do tipo S é significativamente maior que o tempo gasto na resolução das questões do tipo R.

H_1: $\mu PS < \mu PR$ ou $\mu TS > \mu TR$

5.3.3 Variáveis Independentes

A principal variável independente do estudo corresponde ao tipo das expressões OCL presentes em cada questão respondida pelos participantes. Esta variável pode assumir os seguintes valores: S (restrições contendo *OCL smells)* ou R (restrições reestruturadas).

O nível de conhecimento de OCL, a experiência em desenvolvimento de software em geral, em desenvolvimento de software orientado a objetos, bem como a experiência dos participantes com modelos UML também são informações independentes que poderiam afetar de forma indesejável os resultados do estudo.

5.3.4 Variáveis Dependentes

Neste estudo, as variáveis dependentes são o número de pontos e o tempo gasto na resolução das questões que foram apresentadas aos participantes.

A pontuação para cada questão (PQ) por participante é determinada pelo pesquisador a partir da avaliação da solução apresentada pelo participante, correspondendo a um número inteiro entre 0 e 2 pontos. Cada questão consiste em avaliar se uma restrição é violada por um determinado conjunto de instâncias do modelo. Caso a resposta (Sim/Não) esteja correta, e a justificativa (indicação das instâncias que contribuem para uma eventual violação) também esteja correta, ela recebe 2 pontos. Caso a resposta esteja correta, mas a justificativa não englobe todas as instâncias envolvidas na violação, ela é pontuada com o valor 1. Finalmente, caso a resposta esteja incorreta, ou a justificativa não indique nenhuma das instâncias que contribuem para a violação, ela é pontuada com o valor 0.

A partir de PQ, definimos duas variáveis, PS e PR, que correspondem ao somatório de pontos para um participante nas questões dos tipos S e R, respectivamente. Desta forma, PS e PR podem assumir valores entre 0 e 10, uma vez que cada participante respondeu a cinco questões de cada tipo.

O tempo gasto pelo participante na resolução de uma questão é dado pela variável TQ. A partir de TQ, definimos duas variáveis, TS e TR, que correspondem ao somatório do tempo gasto por um participante nas questões dos tipos S e R, respectivamente. O procedimento utilizado para a medição do tempo de resolução de cada questão está descrito na seção 5.4.3.

5.3.5 Seleção dos Participantes

Os participantes foram escolhidos por conveniência. A partir de uma consulta prévia a um universo de cerca de cem pessoas, englobando alunos de mestrado e doutorado em Engenharia de Software da UFRJ, profissionais da indústria que foram

alunos do programa de pós-graduação IS-EXPERT do NCE-UFRJ, além de profissionais de empresas do relacionamento do pesquisador, vinte e duas pessoas se candidataram, voluntariamente, a participar do estudo. A composição do grupo de participantes é apresentada na Tabela 5.1.

Origem	Participantes
IS-EXPERT – turma 2000	1
IS-EXPERT – turma 2003	6
IS-EXPERT – turma 2004	3
IS-EXPERT – turma 2005	6
COPPE-UFRJ (alunos de doutorado que já participaram de projetos na indústria)	2
Profissionais de empresas que não foram alunos dos cursos acima	4

Tabela 5.1– Seleção dos Participantes

Os participantes foram selecionados através de comunicações realizadas por meio de mensagens de correio eletrônico. A Tabela 5.2 apresenta um resumo do perfil dos participantes de acordo com a formação, experiência em desenvolvimento de software, experiência em desenvolvimento orientado a objetos, experiência na utilização de UML e de OCL.

Formação	Doutorando	2
	Mestre	2
	Especialização	18
Experiência em Desenvolvimento	Menos de 1 ano	2
	De 3 a 5 anos	2
	De 5 a 10 anos	5
	Mais de 10 anos	13
Experiência em OO	Apenas fez um curso	1
	Utilizou em projeto de curso	6
	Até 3 projetos	5
	Mais de 3 projetos	10
Experiência em UML	Apenas fez um curso	2
	Utilizou em projeto de curso	6
	Até 3 projetos	10
	Mais de 3 projetos	4
Experiência em OCL	Nenhuma	5
	Apenas fez um curso	10
	Utilizou em projeto de curso	7
	Até 3 projetos	0

Tabela 5.2 – Quadro Resumo dos Participantes

A maior parte dos participantes tem experiência de utilização da UML em projetos na indústria, mas nenhum deles utiliza a OCL no cotidiano do seu ambiente de trabalho. A maioria dos participantes já possuia algum conhecimento de OCL, em função de terem sido alunos de uma disciplina de elaboração de modelos de software com UML e OCL do curso de pós-graduação IS-EXPERT do NCE-UFRJ. Entretanto, nenhum dos participantes tinha experiência prática de utilização da OCL em projetos na indústria. Todos os participantes assinaram um termo de consentimento e preencheram um formulário de caracterização.

Em função de restrições de agenda dos participantes e da localização geográfica dispersa dos mesmos, julgamos que não seria viável reuni-los no mesmo local para que o experimento fosse executado de forma simultânea e presencial. A solução adotada consistiu em realizar o estudo experimental através de reuniões individuais e remotas, utilizando o programa Microsoft Messenger, pois todos os participantes declararam ter ao menos alguma experiência no uso desse programa.

5.3.6 Projeto do Experimento

Com o objetivo de aumentar o número de pontos para a análise estatística, dado o número de participantes, todos os participantes foram submetidos aos dois tratamentos, isto é, todos os participantes analisaram restrições contendo *OCL smells* (tipo S) e expressões OCL reestruturadas (tipo R). Os participantes foram divididos em dois grupos (I e II). Cada participante respondeu a um questionário contendo 10 questões formuladas sobre as restrições. Foram elaborados dois questionários (QI e QII) contendo questões sobre o mesmo modelo de classes, totalizando 20 questões, sendo 10 de cada questionário. Os questionários contêm 5 questões do tipo S e 5 questões do tipo R, e foram formulados de modo a apresentarem restrições com o mesmo grau de dificuldade, tanto global como por tipo de questão, isto é, questões de um determinado tipo (S ou R) do questionário I deveriam apresentar um grau de dificuldade semelhante às questões do mesmo tipo do questionário II. A composição dos dois questionários é descrita em detalhes na seção 5.3.7.

A alocação dos participantes aos questionários QI e QII foi feita considerando-se não somente o conhecimento de OCL dos participantes, medido através da nota atribuída pelo pesquisador a um questionário preliminar (QP) respondido pelos participantes, mas também a origem dos participantes (IS-EXPERT, COPPE-UFRJ,

participantes de empresas que não foram alunos desses programas). Embora admitamos que existam formas mais robustas de se avaliar o conhecimento dos participantes em OCL, a aplicação de um questionário foi aquela que julgamos como mais adequada em função do tempo disponível dos participantes.

Os participantes foram, então, classificados em relação ao seu nível de conhecimento em dois blocos, de acordo com a mediana das notas de QP:

- Alto: participantes com nota acima da mediana;
- Baixo: participantes com nota abaixo da mediana.

Com base nessa classificação, os participantes foram alocados aleatoriamente aos questionários QI e QII, com a restrição de que cada questionário deveria ser respondido por um número o mais próximo possível de participantes de cada bloco (Alto ou Baixo). O mesmo critério foi adotado em relação à origem dos participantes.

Para evitar a influência da ordem de resolução das questões, o participante deveria responder a uma questão por vez, sendo que as questões de cada tipo foram apresentadas de forma intercalada, ou seja, se a primeira questão fosse do tipo S, a questão seguinte seria do tipo R, e assim sucessivamente. O tipo da primeira questão foi definido aleatoriamente para cada participante. A partir daí, o tipo da questão foi sendo invertido até que o participante respondesse à última pergunta.

5.3.7 Instrumentação

Esta seção descreve os instrumentos utilizados neste estudo. Todos estes instrumentos estão disponíveis no seguinte endereço:

http://reuse.cos.ufrj.br/~alexcorr/estudo01.htm.

a) Instruções para o Participante (IP)

Documento contendo instruções sobre as atividades que seriam realizadas pelo participante durante o experimento.

b) Formulário de Caracterização do Participante (CP)

Formulário onde o participante descreve sua experiência e conhecimento em alguns itens relevantes para o estudo, como por exemplo, tempo de experiência em desenvolvimento de software, experiência com UML e OCL.

c) Tutorial OCL

Todos os participantes receberam um tutorial de cerca de 30 páginas, elaborado pelo pesquisador, contendo os principais conceitos da OCL e uma descrição, com exemplos, de todas as operações definidas na biblioteca padrão da linguagem. Esse tutorial foi utilizado como treinamento não assistido em OCL.

d) Questionário Preliminar (QP)

Após estudar o tutorial, cada participante respondeu a um questionário preliminar (QP) contendo dez questões de avaliação de expressões OCL sobre um conjunto de instâncias de um modelo UML. A consulta ao tutorial foi permitida nesta atividade. Esse questionário contém 10 questões envolvendo expressões OCL sobre um único modelo UML, que foram elaboradas com o cuidado de abranger o maior número possível de construções da linguagem. Essas questões contêm expressões OCL que abrangem as construções básicas da linguagem. As expressões presentes nas questões deste questionário preliminar são mais simples do que aquelas presentes nos questionários QI e QII, considerando os dois tipos de questão (R ou S).

e) Modelo UML (M1)

Contém um diagrama de classes referente a um sistema de transações relacionadas com programas de fidelidade de clientes como, por exemplo, os programas de companhias aéreas e de empresas de cartão de crédito. Esse modelo contém doze classes, treze associações e duas generalizações, e foi utilizado para a formulação das questões dos dois questionários (QI e QII).

f) Questão de Aquecimento (QA)

A questão de aquecimento foi elaborada com o propósito de familiarizar o participante com o formato das dez questões que foram objeto de análise. Essa questão foi particularmente importante para que os participantes entendessem o procedimento que deveria ser seguido durante a resolução das dez questões do questionário, de forma a não comprometer a correção e a medição de tempo de resolução, especialmente na primeira questão, devido a problemas de entendimento do procedimento.

g) Questionários QI e QII

Esses questionários contêm questões de avaliação de restrições referentes ao modelo descrito no item *e* desta seção. O número e a dificuldade das questões foram definidos de modo que o questionário pudesse ser respondido, em média, em menos de 60 minutos.

Cada questão desses questionários é composta por três partes:

- Uma restrição descrita em OCL que pode referenciar uma ou mais definições também elaboradas em OCL. Nenhuma das expressões OCL foi acompanhada por descrições em linguagem natural;
- Um diagrama de objetos contendo algumas instâncias e ligações entre instâncias do modelo;
- Uma pergunta solicitando que o participante indique se a restrição é violada pelas instâncias presentes no diagrama de objetos e justifique a sua resposta.

As questões do tipo R de um questionário contêm os mesmos diagramas de objetos presentes nas questões do tipo S do outro questionário. Entretanto, as restrições das questões do tipo R correspondem a versões reestruturadas das restrições presentes nas questões do tipo S do outro questionário. Desta forma, os conjuntos de perguntas e diagramas de objetos são os mesmos nos questionários QI e QII. A diferença está na forma como as expressões OCL utilizadas em cada questão estão estruturadas. Portanto, para cada questão do questionário QI que esteja relacionada a expressões contendo *OCL smells* (tipo S), o questionário QII apresenta a mesma questão elaborada com uma versão reestruturada dessas expressões (tipo R), e vice-versa.

As questões foram elaboradas de forma a expor todos os participantes ao mesmo número de expressões com *OCL smells* de graus de dificuldade semelhantes. Assim, se o questionário QI possui uma questão contendo o *OCL smell X*, o questionário QII também apresenta uma questão contendo o *OCL smell X*, só que, neste caso, envolvendo outras expressões.

A Tabela 5.3 apresenta um quadro resumo da estrutura de cada questionário. Os *OCL smells* presentes em cada questão do tipo S estão descritos na coluna *OCL Smells*. A coluna "*Quest. Relac."* indica o número da questão do outro questionário

que contém uma restrição com *OCL smells* semelhantes. Desta forma, a correspondência entre as questões contendo *OCL smells* similares é dada pelos seguintes pares: (S1-S10); (S2-S5); (S3-S4); (S6-S9) e (S7-S8).

Questionário QI			**Questionário QII**		
Questão	*OCL Smells*	*Quest. relac.*	*Questão*	*OCL Smells*	*Quest. relac.*
S1	Cadeia de implicações	10	R1		
S3	Expressão Prolixa e Conversão de Tipos	4	R3		
S5	Expressão Prolixa e Cadeia Quantificadores Universais	2	R5		
S7	Expressão Prolixa – Contexto Inadequado	8	R7		
S9	Expressão condicional relacionada a tipo	6	R9		
R2			S2	Expressão Prolixa e Cadeia Quantificadores Universais	5
R4			S4	Expressão Prolixa e Conversão de Tipos	3
R6			S6	Expressão condicional relacionada a tipo	9
R8			S8	Expressão Prolixa – Contexto Inadequado	7
R10			S10	Cadeia de implicações	1

Tabela 5.3 – Composição dos questionários QI e QII em relação aos tipos de *OCL smells* presentes

Esses questionários geraram duas fontes de informação para análise: a correção das respostas de cada questão e o tempo gasto pelo participante na resolução de cada questão. Para que fosse possível medir o tempo por questão, as questões foram apresentadas uma por vez para cada participante, e este tinha que respondê-la antes de passar para a próxima questão. A forma como as questões foram apresentadas e a estratégia usada para a medição do tempo são discutidas na seção 5.4.3.

f) Formulário de Avaliação Subjetiva (AS)

Este instrumento é composto por perguntas que visam capturar a avaliação subjetiva dos participantes em relação às questões do questionário respondido (QI ou QII). Os participantes avaliaram a dificuldade de entendimento das expressões OCL

presentes nas questões. Cada questão recebeu uma avaliação na escala de 1 (muito fácil) a 5 (muito difícil). Além disso, cada participante opinou sobre a estrutura das expressões OCL presentes nas questões, sendo que cada questão recebeu uma avaliação de 1 a 5, de acordo com a seguinte escala:

1 - as expressões OCL desta questão poderiam ter sido estruturadas de uma forma melhor (de forma mais simples, por exemplo), mas eu não sei bem como fazê-lo;
2 - as expressões OCL desta questão poderiam ter sido estruturadas de uma forma melhor (de forma mais simples, por exemplo), e eu tenho uma idéia de como fazê-lo;
3 - não tenho opinião sobre as expressões OCL desta questão. Não sei julgar se elas poderiam ter sido estruturadas de uma forma melhor ou se estão adequadamente estruturadas;
4 - as expressões OCL desta questão estão adequadamente estruturadas, mas ainda é possível realizar alguns pequenos aperfeiçoamentos na sua estrutura;
5 - as expressões OCL desta questão estão adequadamente definidas, e não vejo necessidade de modificá-las.

Cada participante foi instruído para descrever as sugestões de modificação na estrutura das expressões presentes nas questões que fossem avaliadas com o número 2 ou 4.

Além dessas informações, o formulário solicitou do participante sua avaliação em relação à organização e à clareza das instruções do experimento, além de solicitar sugestões para o aperfeiçoamento do estudo em futuras replicações do mesmo.

A coleta destas informações teve o propósito de verificar se a dinâmica e os instrumentos utilizados exerceram alguma influência sobre os resultados do estudo, além de complementar a análise dos resultados objetivos obtidos através dos questionários QI e QII.

5.4 Operação do Estudo

5.4.1 Treinamento

Inicialmente, todos os participantes receberam o tutorial em OCL e o questionário preliminar (QP). Os objetivos deste questionário foram:

- possibilitar que os participantes tivessem conhecimento teórico sobre OCL suficiente para resolver as questões utilizadas no estudo;
- permitir que os participantes aplicassem os conhecimentos teóricos descritos no tutorial na resolução de questões envolvendo a avaliação de expressões e restrições sobre um espaço hipotético de instâncias de um modelo;
- identificar possíveis disparidades de conhecimento em UML/OCL, de forma a orientar a seleção dos participantes e dos respectivos materiais que eles utilizariam no experimento.

Cada participante teve o prazo de uma semana para estudar o tutorial OCL, e enviar as respostas do questionário QP para o pesquisador. Uma vez enviadas as respostas, cada participante recebeu o gabarito do questionário contendo a explicação detalhada da resolução das questões, de forma a consolidar o seu aprendizado.

5.4.2 Alocação dos Participantes aos Questionários QI e QII

A alocação dos participantes aos questionários QI e QII foi feita considerando-se não somente o conhecimento de OCL dos participantes, medido através da nota atribuída pelo pesquisador ao questionário preliminar (QP), como também a origem dos participantes (programa IS-EXPERT, COPPE-UFRJ, profissionais de desenvolvimento que não foram alunos desses programas).

A Tabela 5.4 apresenta um resumo da distribuição dos participantes pelos dois questionários de acordo com as notas no questionário QP.

Questionário QI		**Questionário QII**	
Alto	Baixo	Alto	Baixo
6	5	5	6

Tabela 5.4 - Distribuição dos participantes nos questionários de acordo com a avaliação QP

A Tabela 5.5 apresenta um resumo da distribuição dos participantes pelos dois questionários de acordo com a sua origem (programa IS-EXPERT, COPPE, outros).

Questionário QI			**Questionário QII**		
IS-EXPERT	COPPE	Outros	IS-EXPERT	COPPE	Outros
8	1	2	8	1	2

Tabela 5.5 - Distribuição dos participantes nos questionários de acordo com a origem

5.4.3 Execução

A execução do experimento foi realizada em cinco dias. Uma vez que o estudo foi executado através de reuniões individuais remotas, utilizando o programa Microsoft Messenger, vários cuidados foram tomados para evitar interferências indesejáveis tanto do ambiente computacional como do ambiente externo.

No início da reunião, o participante descreveu o ambiente onde ele se encontrava e, em particular, o nível de ruído e a possibilidade de ocorrerem interrupções não esperadas. Uma vez que os participantes escolheram o local e o horário de sua maior conveniência, nenhum participante relatou estar em um ambiente que pudesse interferir negativamente no seu desempenho. Cada participante recebeu recomendações detalhadas sobre os procedimentos a serem seguidos e sobre a importância de não permitir interrupções durante a resolução de uma questão. Admitiu-se, entretanto, interrupção entre a resolução de uma questão e outra, mas isto raramente foi necessário. Para evitar a ocorrência de interrupções indesejáveis através do próprio programa Microsoft Messenger, cada participante recebeu instruções para definir uma mensagem para os demais usuários não o chamarem durante o experimento.

Após as instruções iniciais, o participante recebeu um arquivo contendo o modelo UML (M1), que foi utilizado nas restrições presentes nos questionários aplicados. Cada participante teve cinco minutos para estudá-lo.

Em seguida, o participante recebeu a questão de aquecimento (QA), que teve como objetivo instrui-lo sobre o procedimento que deveria ser seguido na resolução de cada questão, além de familiarizá-lo com a estrutura das questões que seriam apresentadas na seqüência, bem como na forma de estruturação da sua resposta. O procedimento adotado para a resolução de cada questão seguiu o seguinte protocolo:

- Pesquisador envia a seguinte mensagem para o participante: “Vendo questão?”;
- Pesquisador envia o arquivo correspondente à questão para o participante;
- Participante aceita o recebimento do arquivo;
- Participante abre o arquivo com a questão;
- Assim que o participante estiver vendo a questão aberta na tela, ele envia a seguinte mensagem para o pesquisador: “ok vendo”;
- Pesquisador dispara a contagem de tempo;
- Participante resolve a questão;
- Participante envia a resposta através de uma mensagem dividida em duas partes: Sim/Não (se há ou não violação da restrição nas instâncias referentes à questão); justificativa (que deve indicar explicitamente as instâncias relacionadas com a violação);
- Pesquisador fecha a contagem de tempo para a questão.

Esse protocolo foi seguido na resolução da questão de aquecimento, e o pesquisador enfatizou, com todos os participantes, a importância de que ele fosse corretamente seguido nas questões do questionário que viria na seqüência.

Cada participante recebeu uma questão por vez, seguindo o questionário ao qual ele havia sido previamente alocado. Somente após receber a resposta de uma questão, o pesquisador avançava para a próxima questão. Os participantes puderam recorrer ao material do treinamento para responder as questões do questionário.

Após a aplicação das questões do questionário QI ou QII, o participante recebeu o formulário de avaliação subjetiva (AS) e teve 45 minutos para respondê-lo, finalizando a sua participação no experimento.

5.5 Resultados e Análise

5.5.1 Análise dos Instrumentos

A primeira avaliação realizada foi relacionada aos instrumentos utilizados no experimento. De forma a guiar as análises posteriores, queríamos avaliar se os questionários exerceram algum tipo de influência sobre os resultados, já que eles tinham sido projetados para fazer com que os participantes dos dois grupos passassem por experiências similares.

Para esta avaliação, utilizamos o teste de análise de variância (ANOVA), com um nível de significância (α) de 0.05 (WOHLIN et al., 2000), sobre o seguinte conjunto de dados:

- os pontos obtidos pelos participantes nos dois questionários;
- o tempo gasto pelos participantes na resolução dos dois questionários;
- a avaliação subjetiva da dificuldade das expressões dos dois questionários.

Análise dos pontos obtidos nos questionários QI e QII

Em relação aos pontos obtidos pelos participantes nos dois questionários, buscamos responder as perguntas a seguir:

- **P1**: o número médio de pontos obtidos pelos participantes que responderam o questionário QI foi semelhante ao número médio de pontos obtidos pelos participantes que responderam o questionário QII?
- **P2**: o número médio de pontos obtidos pelos participantes nas questões de um tipo (S ou R) do questionário QI foi semelhante ao número médio de pontos obtidos nas questões do mesmo tipo do questionário QII?

Análise do tempo gasto nos questionários QI e QII

Em relação aos tempo gasto pelos participantes nos dois questionários, buscamos responder as perguntas a seguir:

- **P3**: o tempo médio gasto pelos participantes que responderam o questionário QI foi semelhante ao tempo médio gasto pelos participantes que responderam o questionário QII?

- **P4**: o tempo médio gasto pelos participantes nas questões de um tipo (S ou R) do questionário QI foi semelhante ao tempo médio gasto nas questões do mesmo tipo do questionário QII?

Análise da avaliação subjetiva da dificuldade dos questionários QI e QII

Utilizando a avaliação subjetiva fornecida pelos participantes para cada questão no formulário AS, buscamos responder as questões a seguir:

- **P5**: a dificuldade das questões do questionário QI foi semelhante à dificuldade das questões do questionário QII, segundo o julgamento subjetivo dos participantes?
- **P6**: a dificuldade das questões de um tipo (S ou R) do questionário QI foi semelhante à dificuldade das questões do mesmo tipo do questionário QII, segundo o julgamento subjetivo dos participantes?

A Tabela 5.6 apresenta os resultados da análise correspondente à pergunta P1. De acordo com esses resultados, não houve diferença significativa entre a pontuação média por participante nos dois questionários, uma vez que F (ANOVA) < FCRIT. Portanto, a pergunta P1 é respondida afirmativamente.

Questionário	N	SUM QUAD.	MÉDIA QUAD.	Média de Pontos por Participante
QI	110	368	321	17,09
QII	110	373	332	17,36
F (ANOVA)	0,10	FCRIT	3,88	

Tabela 5.6 - Comparação dos pontos obtidos nos questionários QI e QII

De forma análoga, os resultados obtidos com as análises estatísticas dos instrumentos, tanto do ponto de vista quantitativo quanto do ponto de vista qualitativo, indicam que todas as demais perguntas (P2 a P6) são respondidas afirmativamente. Portanto, os resultados dos participantes nos dois questionários foram analisados em conjunto, visto que não houve diferença significativa de um questionário para outro.

5.5.2 Análise dos Resultados quanto à Correção do Entendimento

a) Estatística Descritiva

A Tabela 5.7 apresenta um quadro resumo dos pontos obtidos por cada participante nas questões do tipo S, do tipo R e no questionário como um todo.

ID	Pontos (Tipo S)	Pontos (Tipo R)	Total
1	10	10	20
2	5	10	15
3	4	9	13
4	7	7	14
5	7	10	17
6	6	10	16
7	9	8	17
8	8	10	18
9	9	10	19
10	7	8	15
11	10	8	18

ID	Pontos (Tipo S)	Pontos (Tipo R)	Total
12	8	10	18
13	9	10	19
14	7	9	16
15	8	10	18
16	8	8	16
17	10	10	20
18	10	10	20
19	10	10	20
20	7	8	15
21	8	10	18
22	8	9	17

Tabela 5.7 – Pontos obtidos pelos participantes nos questionários

A Tabela 5.7 apresenta as médias dos pontos obtidos no total e por tipo de questão. As questões do tipo R apresentaram uma média maior e um desvio padrão menor, se comparadas com as questões do tipo S. Cerca de 60% dos participantes acertaram todas as questões do tipo R, enquanto que apenas 22% dos participantes acertaram todas as questões do tipo S. Apenas dois participantes acertaram mais questões do tipo S do que do tipo R (id = 7 e 11).

ID	Tipo S	Tipo R	Geral
Média	7,95	9,27	17,22
Desvio Padrão	1,65	0,98	2,04
% com acerto máximo	22%	59%	18%

Tabela 5.8 - Estatísticas descritivas de pontos nos questionários

O gráfico da Figura 5.1 apresenta o percentual de acertos por questão. Nove questões tiveram índice de 100% de acerto, sendo seis do tipo R e três do tipo S. Os quatro piores índices de acerto correspondem a questões do tipo S (S5, S10, S6 e S9).

Figura 5.1 – Percentual de acertos por questão

b) Teste Paramétrico

O teste de análise de variância (ANOVA) com um nível de significância (α) de 0.05 foi aplicado aos pontos obtidos nas questões do tipo R e do tipo S. Na população analisada, o resultado (Tabela 5.9) rejeitou a hipótese de que a média dos pontos obtidos nas questões do tipo R (μPR) não seria significativamente diferente da média dos pontos obtidos nas questões do tipo S (μPS), em favor da hipótese alternativa H_1: μPS < μPR.

Tipo Questão	N	SUM QUAD.	MÉDIA QUAD.	Média de Pontos por Participante
S	110	339	278	7,95
R	110	402	378	9,27
F (ANOVA)	9,89	FCRIT	6,75	

Tabela 5.9 - Análise de Variância dos pontos obtidos por tipo de questão (R e S)

Desta forma, tanto o teste paramétrico como a análise da estatística descritiva dos acertos das questões indicam que, na população analisada, a presença de *OCL smells* em expressões OCL pode afetar negativamente a correta compreensão de restrições.

5.5.3 Análise dos Resultados quanto ao Tempo

a) Estatística Descritiva

A Tabela 5.10 apresenta um quadro resumo do tempo gasto por cada participante nas questões do tipo S, do tipo R e no questionário como um todo. A grande maioria dos participantes levou mais tempo resolvendo as questões do tipo S do que as do tipo R. Apenas quatro participantes não apresentaram uma diferença significativa nos tempos entre os tipos de questão (1, 3, 18 e 20). Os participantes 1 e 18 obtiveram pontuação máxima nos questionários, enquanto que os participantes 3 e 20 obtiveram pontuações dentre as mais baixas. Entretanto, outros participantes que obtiveram pontuação máxima (17 e 19) apresentaram uma diferença significativa entre os tipos de questão.

ID	Tempo (Tipo S)	Tempo (Tipo R)	Total
1	10:50	12:00	22:50
2	41:30	35:20	1:16:50
3	20:50	20:30	41:20
4	27:40	22:40	50:20
5	35:00	18:30	53:30
6	44:20	26:40	1:11:00
7	32:30	25:00	57:30
8	21:10	15:10	36:20
9	34:00	30:20	1:04:20
10	32:30	25:00	57:30
11	21:10	15:10	36:20

ID	Tempo (Tipo S)	Tempo (Tipo R)	Total
12	28:00	18:30	46:30
13	24:20	20:50	45:10
14	32:10	23:40	55:50
15	30:20	19:30	49:50
16	21:20	12:30	33:50
17	32:00	22:50	54:50
18	21:20	21:10	42:30
19	18:20	14:30	32:50
20	14:10	15:00	29:10
21	33:40	20:30	54:10
22	16:10	12:10	28:20

Tabela 5.10 - Tempo gasto pelos participantes nos questionários

A Tabela 5.11 apresenta as médias do tempo de resolução do conjunto de questões de cada tipo, bem como a média geral de resolução de um questionário. Além disso, os tempos mínimo e máximo para a resolução de um questionário e para a resolução das questões de cada tipo também são apresentadas. As questões do tipo S apresentaram uma média de tempo para a resolução das questões do tipo S superior à média de tempo para as questões do tipo R. O desvio padrão para o tempo das questões do tipo S também é maior do que o desvio padrão para as questões do tipo R.

ID	Tipo S	Tipo R	Geral
Média	26:12	20:00	46:12
Desvio Padrão	8:56	5:55	14:11
Mínimo	10:50	12:00	22:50
Máximo	44:20	35:20	1:16:50

Tabela 5.11 – Estatísticas descritivas do tempo gasto nos questionários

O gráfico ilustrado na Figura 5.2 apresenta o tempo médio gasto em cada questão. Cada questão está associada com duas barras no gráfico, sendo que a barra mais à esquerda corresponde a uma questão do tipo S, enquanto que a barra da direita corresponde a uma questão do tipo R. Em 90% das questões, o tempo gasto na resolução da questão do tipo S foi maior do que na mesma questão do tipo R.

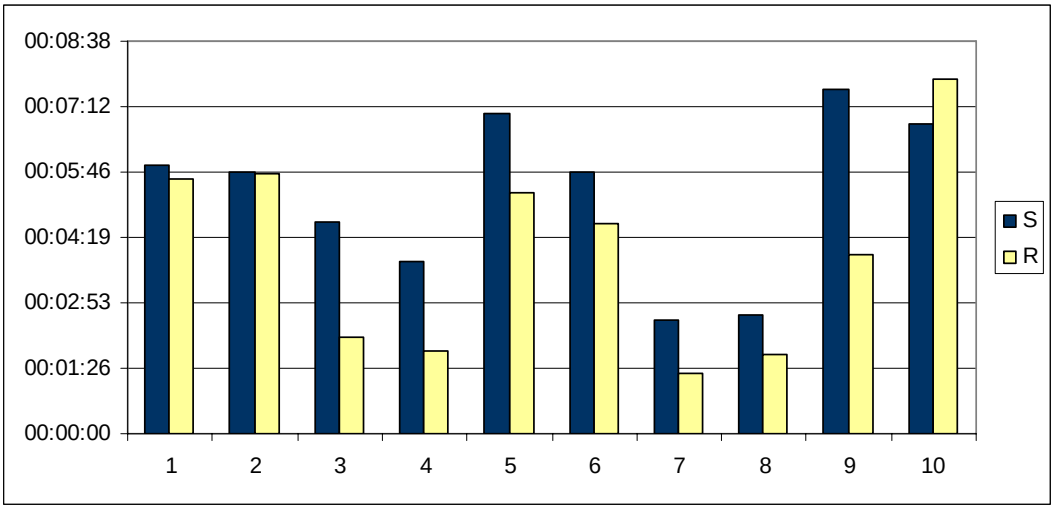


Figura 5.2 – Tempos por questão / tipo

b) Teste Paramétrico

O teste de análise de variância (ANOVA) com um nível de significância (α) de 0.05 foi aplicado aos tempos utilizados para responder as questões do tipo R e do tipo S. Antes, porém, os pontos extremos foram eliminados, utilizando-se como critério os tempos de resolução de questão que ficaram 1,5 desvio padrão acima ou abaixo da média, o que resultou na eliminação de 12 pontos.

Na população analisada, o resultado rejeitou a hipótese de que o tempo médio necessário para resolver as questões do tipo R (μTR) não seria diferente do tempo

médio necessário para resolver as questões do tipo S (μTS), em favor da hipótese alternativa H_1: μTS > μTR.

Tipo Questão	N	SUM QUAD.	MÉDIA QUAD.	Média de Tempo por Questão
S	105	13.236.000	10.047.146,67	05:14
R	103	8.525.400	5.942.403,88	04:00
F (ANOVA)	8,87	FCRIT	3,89	

Tabela 5.12 - Análise de Variância dos tempos por tipo de questão (R e S)

5.5.4 Análise da Avaliação Subjetiva da Dificuldade das Questões

Os dados do formulário de avaliação subjetiva foram utilizados para analisar a dificuldade percebida pelos participantes na resolução das questões dos dois tipos. Em paticular, estávamos interessados em investigar se houve diferença na percepção de dificuldade dos participantes em relação ao tipos de questão.

a) Estatística Descritiva

A Figura 5.3 apresenta um gráfico comparativo do nível de dificuldade por tipo de questão, de acordo com o julgamento dos participantes.

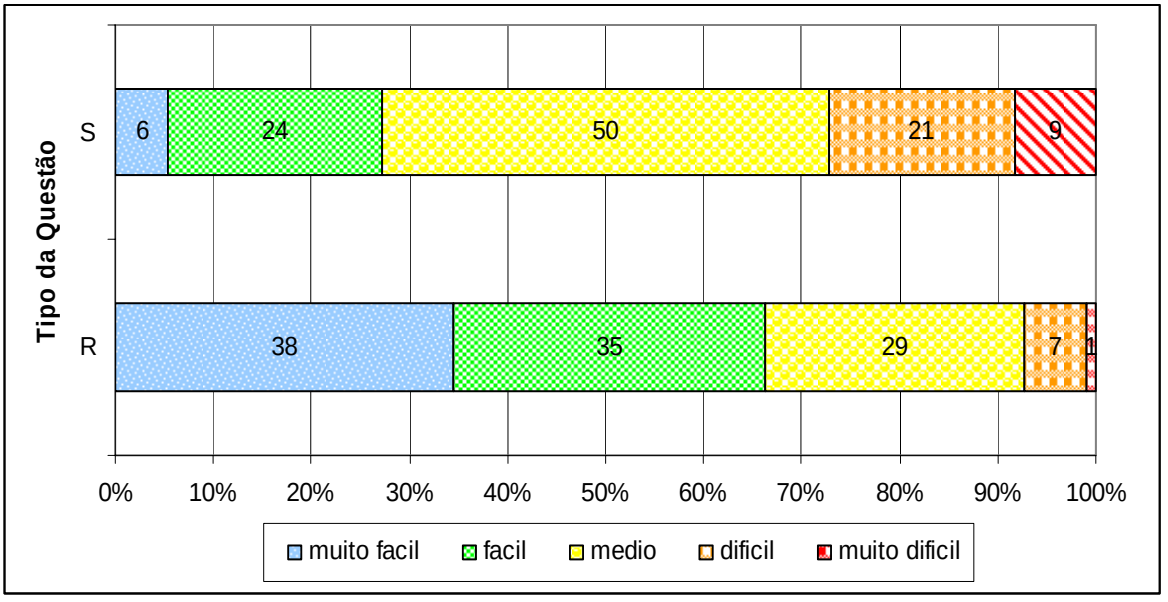


Figura 5.3 – Comparação do nível de dificuldade por tipo de questão

Os valores *muito fácil* ou *fácil* corresponderam a mais de 60% dos julgamentos atribuídos a questões do tipo R, contra menos de 30% dos julgamentos atribuídos a questões do tipo S. Além disso, os valores *difícil* e *muito difícil* corresponderam a quase 30% dos julgamentos atribuídos a questões do tipo S, contra menos de 10% do tipo R.

O gráfico ilustrado na Figura 5.4 apresenta a frequência de cada resposta para as questões do tipo R e S. As questões de 3 a 9 apresentaram uma clara diferença no julgamento de dificuldade, sendo as questões R julgadas com mais frequência como *fáceis* ou *muito fáceis*. As questões 1 e 10 foram julgadas de modo muito similar. Ambas correspondem a restrições com a presença do *OCL Smell Cadeia de Implicações*.

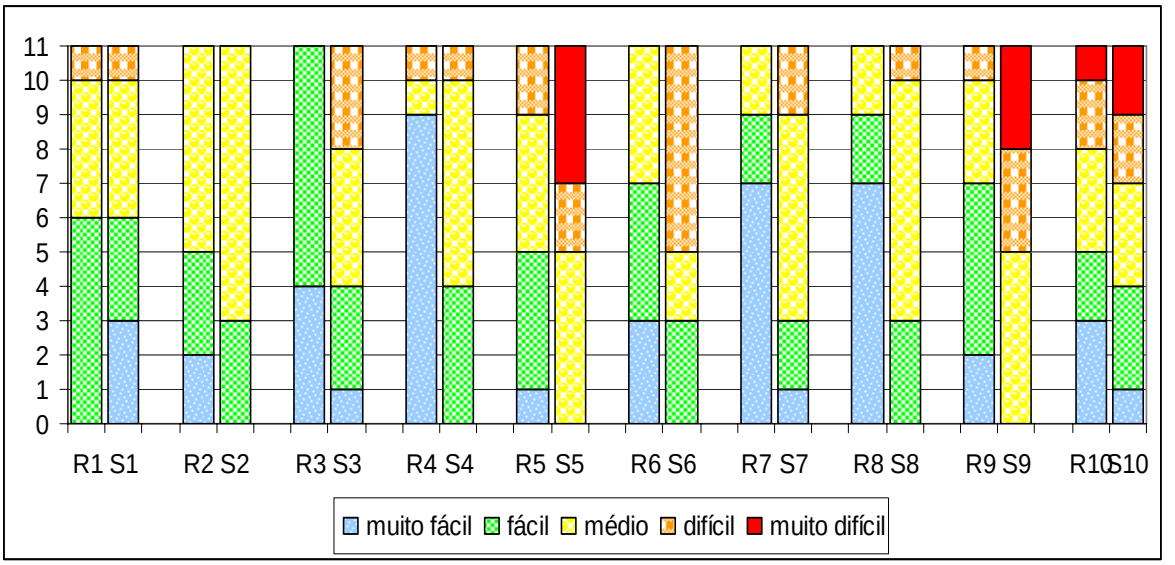


Figura 5.4 – Comparação do nível de dificuldade por questão

b) Correlação Dificuldade x Pontos

Analisamos a possibilidade de uma correlação entre o grau de dificuldade percebida pelos participantes e a pontuação média obtida por eles em cada questão. O grau de dificuldade foi obtido a partir de um mapeamento dos julgamentos da escala ordinal para a escala intervalar, através de uma função monotônica crescente que preserva a ordem dos seus elementos. O resultado (-0,71) indica uma forte correlação negativa entre esses dois elementos, ou seja, quanto maior a dificuldade percebida pelos participantes em uma questão, menor o número médio de pontos obtidos na sua

resolução. A Figura 5.5 apresenta o gráfico de dispersão entre a pontuação média e o grau de dificuldade das 22 questões analisadas.

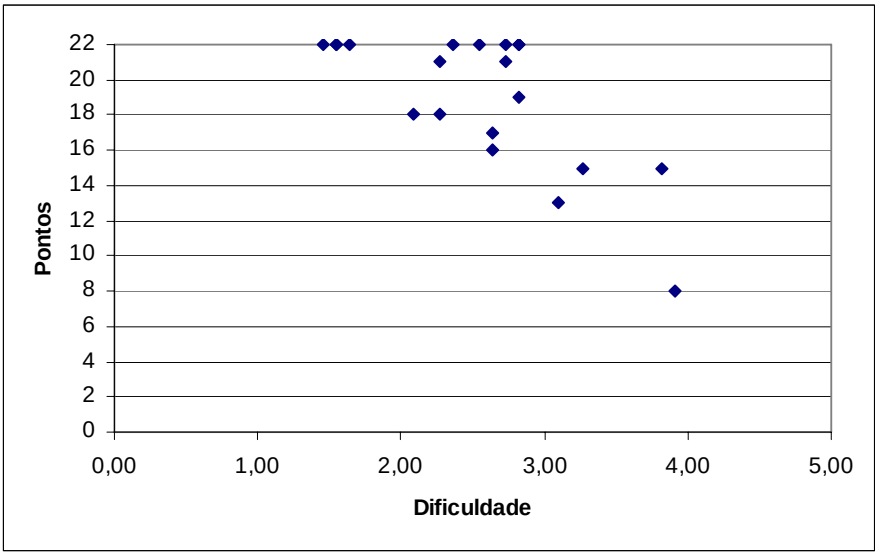


Figura 5.5 – Correlação Dificuldade x Pontos

c) Correlação Dificuldade x Tempo

Analisamos também a possibilidade de uma correlação entre o grau de dificuldade percebida pelos participantes e o tempo médio gasto na resolução de cada questão. O resultado (0,73) indica uma forte correlação entre esses dois elementos, ou seja, quanto maior a dificuldade percebida pelos participantes em uma questão, maior o tempo médio gasto por eles na sua resolução. A Figura 5.6 apresenta o gráfico de dispersão entre o tempo médio de resolução e o grau de dificuldade das 22 questões analisadas.

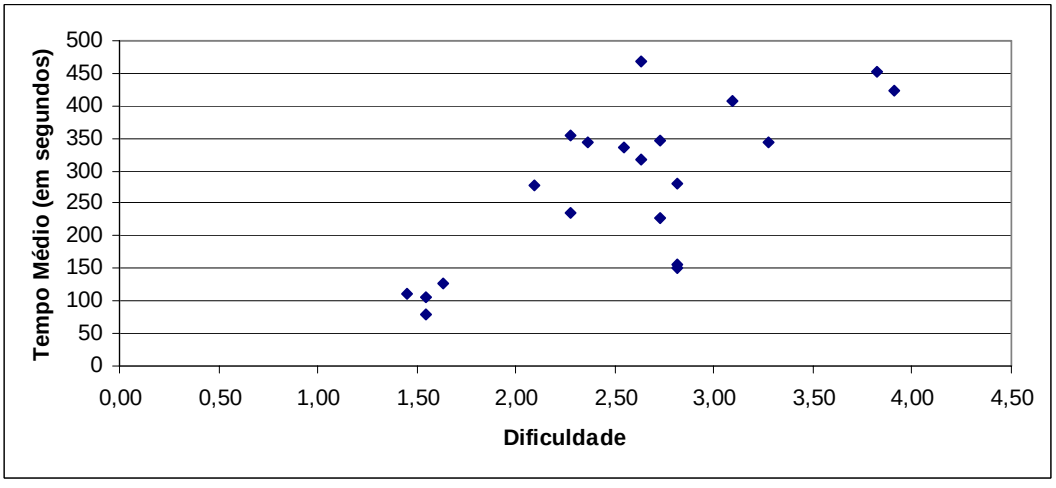


Figura 5.6 - Correlação Dificuldade x Tempo

5.5.5 Análise da Avaliação Subjetiva da Qualidade das Expressões

Os participantes julgaram a qualidade da estrutura das expressões OCL presentes em cada questão do questionário. Os resultados dessa avaliação são apresentados nesta seção.

a) Questões do Tipo S

A Figura 5.7 apresenta a distribuição percentual das avaliações da estrutura das questões do tipo S, de acordo com os cinco julgamentos possíveis. Apenas 19% dos participantes julgaram que as expressões das questões do tipo S poderiam, de fato, ser estruturadas de uma forma melhor e indicaram alternativas para tal. Apenas dois participantes (1 e 19) fizeram sugestões que levariam as expressões para um nível de estruturação semelhante ao das questões do tipo R correspondentes. Vale ressaltar, ainda, que das cinco questões do tipo S, o participante 1 sugeriu mudanças em apenas duas, e o participante 19 sugeriu mudanças em três.

17% dos participantes detectaram que a estrutura das expressões das questões do tipo S poderiam ser melhoradas, mas declararam não saber como fazê-lo. Por outro lado, 44% dos participantes julgaram a estrutura das expressões como boas ou excelentes.

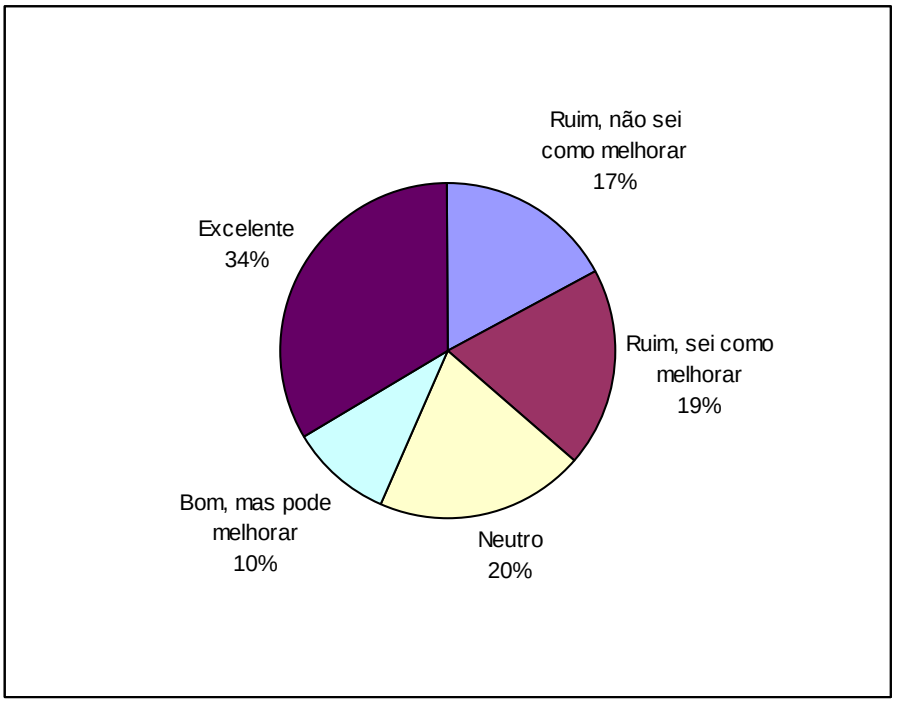


Figura 5.7 – Julgamento Subjetivo da Estrutura das Questões do Tipo S

b) Questões do Tipo R

A Figura 5.8 apresenta a distribuição percentual das avaliações da estrutura das questões do tipo R, de acordo com os cinco julgamentos possíveis. Cerca de 80% dos participantes julgaram a estrutura dessas questões como boas ou excelentes. Menos de 4% dos participantes julgaram a estrutura dessas questões como deficientes.

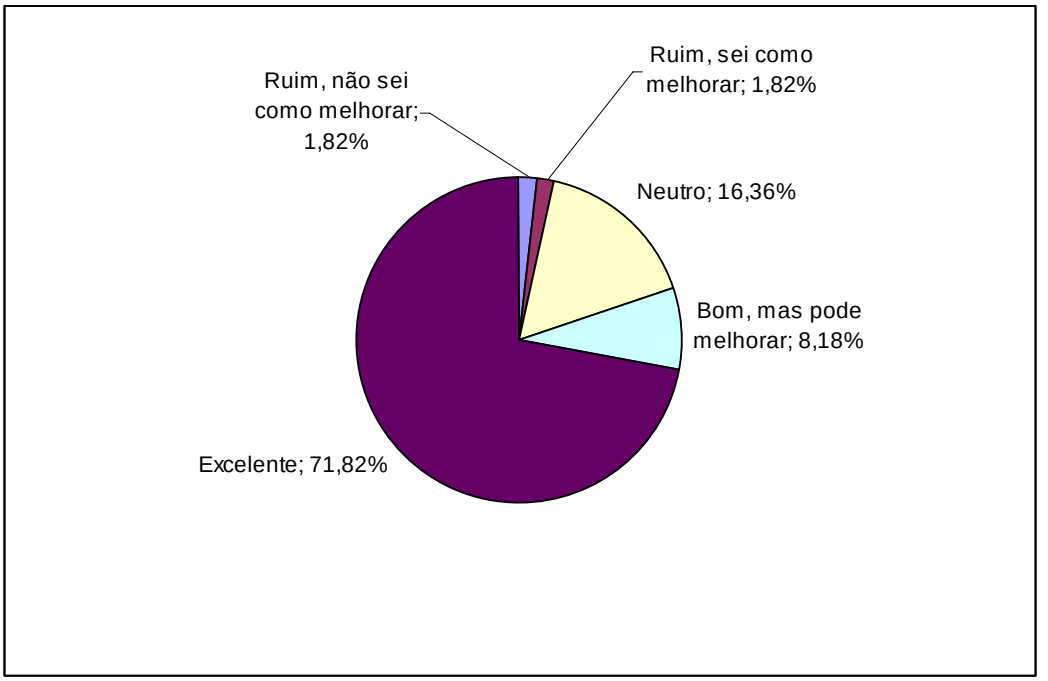


Figura 5.8 – Julgamento Subjetivo da Estrutura das Questões do Tipo R

Vale ressaltar que quatro questões do tipo R (R2, R5, R6 e R9) são versões reestruturadas das questões do tipo S de mesmo número, mas que representam reestruturações parciais, isto é, embora a estrutura das suas expressões OCL seja significativamente diferente daquela presente nas expressões das respectivas questões do tipo S, elas ainda podem ser aperfeiçoadas. Dos julgamentos referentes a essas quatro questões (11 participantes x 4 questões = 44 julgamentos), apenas cinco indicaram que as expressões estavam adequadas, mas que poderiam ser aperfeiçoadas. Apenas um participante (19) indicou corretamente a mudança que poderia ser feita para simplificar as expressões na questão R9. As demais sugestões foram apenas referentes à apresentação das expressões, como por exemplo, modificar a identação ou colocar algumas palavras em negrito.

Os resultados das avaliações das questões tanto do tipo S como do tipo R indicam que boa parte dos participantes conseguiu perceber a diferença na qualidade da estruturação das expressões presentes nas questões, embora a grande maioria não tenha conseguido indicar corretamente as estratégias mais adequadas de reestruturação para as expressões com estrutura deficiente.

5.6 Avaliação da Validade dos Resultados

Esta seção discute os diferentes tipos de ameaça à validade deste experimento em ordem decrescente de prioridade: interna, externa, construção e conclusão (WOHLIN et al., 2000).

5.6.1 Validade Interna

A validade interna de um estudo é definida como a capacidade de um novo estudo replicar o comportamento do estudo atual com os mesmos participantes e objetos com que ele foi realizado.

Uma limitação deste estudo reside no fato dos participantes terem participado do experimento em diferentes locais, dias e horários. Desta forma, como as circunstâncias não eram as mesmas, existe um risco delas terem exercido uma influência inesperada nos resultados. Entretanto, julgamos que esse risco tenha sido minimizado, em função de todos os participantes terem sido submetidos aos mesmos tratamentos, e também pela estratégia adotada de apresentação intercalada dos tratamentos (S e R) durante a sessão. Conforme descrito na seção 5.4.3, vários cuidados foram tomados para minimizar eventuais efeitos causados por influência do ambiente, uma vez que o experimento foi conduzido através de reuniões à distância por meio eletrônico.

Por outro lado, a utilização de um meio eletrônico para a execução do experimento permitiu a participação de um número de pessoas em um espaço de tempo que não seria alcançado caso o experimento fosse executado em um único local e horário, uma vez que os participantes são profissionais que trabalham em diferentes organizações do Rio de Janeiro e, muitas vezes, encontram-se fora da cidade por questões ligadas ao trabalho. Neste experimento, por exemplo, um participante encontrava-se em Maceió e outro em São Paulo quando participaram da reunião eletrônica.

Como as reuniões ocorreram em um espaço de tempo de 5 dias, existe a possibilidade de que tenha havido comunicação entre os participantes sobre as atividades e instrumentos do experimento. Algumas medidas foram tomadas para minimizar esse risco. Em primeiro lugar, apenas o pesquisador tinha conhecimento da relação de pessoas que participariam do experimento e da respectiva agenda de reuniões. Em segundo lugar, as pessoas pertencentes ao mesmo círculo de relacionamento, como por exemplo, alunos da mesma turma do curso IS-EXPERT, foram agendadas para o mesmo dia, minimizando a possibilidade de troca de informações. Além disso, o pesquisador solicitou explicitamente a colaboração dos participantes no sentido de não trocarem informações de qualquer natureza sobre o experimento. Pelos resultados obtidos e pela forma com que as reuniões transcorreram, acreditamos que não tenha ocorrido troca de informações entre os participantes a ponto de influenciar os resultados.

Nenhum dos participantes que responderam ao questionário preliminar utilizado para a divisão dos participantes em grupos deixou de participar do experimento até o seu final. Os participantes foram selecionados a partir de pessoas que voluntariamente se candidataram para o experimento. A maior parte delas demonstrou interesse em utilizar ou aprender mais sobre a OCL. Entretanto, como o estudo não tinha como objetivo fazer qualquer tipo de comparação da OCL com outras linguagens, e considerando que o público alvo das técnicas apresentadas nesta tese corresponde a usuários da OCL, acreditamos que esse fato não tenha exercido influência significativa nos resultados.

5.6.2 Validade Externa

A validade externa do estudo mede sua capacidade de refletir o mesmo comportamento em outros grupos de participantes e profissionais da indústria, ou seja, em outros grupos além daquele em que o estudo foi aplicado.

Como na maioria dos experimentos acadêmicos, aqui surge a questão referente à representatividade dos participantes em relação à população de desenvolvedores de software. Tentamos, considerando as diversas limitações de agenda e interesse das pessoas, envolver um número de pessoas com formação e experiência diversas. Apesar da maioria (70/%) ter em comum o fato de ter participado do mesmo curso de pós-

graduação, elas são graduadas em diferentes instituições e possuem um histórico de trabalho em diferentes empresas e em diferentes tipos de sistemas.

Outra questão, inerente a experimentos controlados efetuados com profissionais da indústria, reside no tamanho e na complexidade do modelo e das restrições utilizadas no estudo. Não podemos afirmar que os resultados obtidos neste experimento ocorreriam de forma semelhante em modelos de sistemas reais, maiores e mais complexos. Encaramos este experimento como um primeiro passo que foi dado antes de investirmos na realização de estudos mais aprofundados em projetos reais.

5.6.3 Validade da Construção

A validade de construção do estudo se refere à relação entre os instrumentos e participantes do estudo e a teoria que está sendo estudada.

O experimento foi projetado de forma que todos os participantes recebessem os mesmos tratamentos, respondendo a perguntas de dois questionários. Os participantes foram divididos aleatoriamente em dois grupos, seguindo uma distribuição que considerou a avaliação do seu conhecimento prévio, obtida através de um questionário preliminar, e a sua origem (curso de pós-graduação IS-EXPERT, curso de pós-graduação da COPPE-UFRJ, profissionais de empresas sem relação com esses cursos). A distribuição dos participantes e a elaboração dos questionários foram cuidadosamente realizadas com o objetivo de fazer com que os participantes passassem por experiências similares e comparáveis. Conforme a análise dos instrumentos discutida na seção 5.4.1, os resultados indicam que este objetivo foi atingido pelo projeto do experimento.

As questões do tipo S foram geradas de forma a apresentarem estruturas presentes em especificações reais, como aquelas referenciadas nos capítulos 3 e 4 desta tese. Não podemos afirmar que a estratégia utilizada neste estudo corresponde à melhor forma de aferição da correção do entendimento de uma restrição. Entretanto, todas as questões foram formuladas seguindo a mesma estrutura, de forma a minimizar a possibilidade de gerar mais um fator de influência. Seguimos uma estratégia similar àquelas utilizadas em outros estudos experimentais que visaram avaliar algum aspecto ligado ao entendimento de programas ou especificações (FINNEY et al., 1999), (SNOOK e HARRISON, 2001)e (BRIAND et al., 2005).

Em relação às ameaças ligadas a fatores sociais, todos os participantes foram informados de que não haveria qualquer tipo de competição ou premiação, e que os seus resultados seriam mantidos em sigilo. Os instrumentos foram estruturados de forma a minimizar a possibilidade dos participantes perceberem ou adivinharem as hipóteses do estudo.

5.6.4 Validade da Conclusão

A validade da conclusão do estudo mede a relação entre os tratamentos e os resultados, determinando a capacidade do estudo em gerar alguma conclusão.

Tentamos dar confiabilidade aos resultados deste estudo através da utilização de medidas objetivas e testes estatísticos paramétricos, além da utilização das avaliações subjetivas para apoiar e explicar os resultados quantitativos. Embora o número de participantes (22) seja relativamente pequeno, tentamos gerar um número de pontos de avaliação razoável ao submeter todos os participantes aos dois tratamentos. Além disso, a divisão dos participantes pelos questionários, os cuidados na elaboração dos instrumentos e da coleta de dados, especialmente do tempo de resposta, e os testes estatísticos realizados tentaram minimizar as ameaças à validade das conclusões deste estudo.

5.7 Lições Aprendidas

Um dos maiores desafios deste estudo experimental consistiu na utilização de reuniões eletrônicas para a interação entre o pesquisador e os participantes. Se por um lado, o controle sobre o ambiente pode ter ficado comprometido, por outro, esta forma de execução deu maior flexibilidade de local e horário para os participantes. Em estudos que envolvam medição de tempo, como foi o caso deste estudo experimental, o comportamento de um participante (concluir rapidamente uma atividade, por exemplo) pode exercer influência no comportamento dos demais. As reuniões individuais remotas permitiram que os participantes não sofressem esse tipo de influência, ao mesmo tempo em que possibilitaram a execução simultânea com múltiplos participantes. Neste estudo, restringimos ao máximo de 3 participantes simultâneos, com o objetivo de não comprometer a qualidade da coleta do tempo gasto pelos participantes em cada questão, nem aumentar o tempo de espera dos participantes entre as questões.

A execução de um projeto piloto com um voluntário foi essencial para a geração adequada dos instrumentos e das instruções que deveriam ser seguidas pelos participantes. A primeira versão do estudo apresentou vários problemas. Em primeiro lugar, as questões eram grandes demais, o que forçou o participante a consumir muito tempo rolando o conteúdo de uma questão para frente e para trás, o que poderia, além de influenciar negativamente na avaliação do tempo de resolução das questões, aumentar sobremaneira o tempo total exigido dos participantes. Para agilizar o processo de resolução, e evitar possíveis influências nos resultados, decidimos simplificar as questões, de forma que todas pudessem ser visualizadas completamente na tela, sem necessidade de rolagem do texto.

Uma outra questão relevante está relacionada ao controle do ambiente. Em uma reunião eletrônica onde os participantes não estejam no mesmo local e, principalmente, em estudos onde o tempo seja um fator relevante, é fundamental caracterizar o ambiente em relação às possibilidades de interrupção (telefone, pessoas, programas de mensagem instantânea) e ao nível de ruído do local. Em função da experiência obtida no piloto, onde o participante relatou a ocorrência de várias interferências externas, procuramos estabelecer um protocolo de comunicação adequado para capturar, da forma mais fidedigna possível, o tempo de execução do estudo.

A partir da experiência obtida neste estudo, consideramos que uma possível alternativa para a obtenção de um nível maior de controle sobre o ambiente seria a utilização de uma câmera, que permitisse a visualização do participante e de suas ações em relação ao computador, além de um software de reunião que oferecesse funções de compartilhamento de tela. Com essas ferramentas, acreditamos que o pesquisador possa acompanhar as ações que o participante esteja realizando em um determinado momento, e ter maior controle sobre o ambiente. Entretanto, devemos considerar que esta alternativa pode restringir o universo de participantes, em função das novas exigências de hardware e software.

A organização do estudo e a clareza das instruções foram avaliadas, quase unanimemente, no seu grau máximo. Com relação à forma eletrônica de realização do experimento, apenas um participante declarou que se sentiria mais confortável se as questões estivessem impressas. Acreditamos que um fator que colaborou para esta avaliação foi o fato das questões terem sido elaboradas de forma que pudessem ser visualizadas confortavelmente em um monitor de 14 polegadas.

5.8 Considerações Finais

Neste capítulo, apresentamos o planejamento, a execução e a análise dos resultados de um estudo experimental que visou avaliar se a compreensão de restrições especificadas em OCL pode ser afetada pela estrutura das expressões que a compõem. Os participantes deste estudo responderam a dez questões onde a metade consistia de restrições contendo alguns dos *OCL smells* descritos nesta tese.

Apesar da reduzida população deste estudo (22 participantes), os resultados dos testes estatísticos aplicados indicam que, no universo analisado, a estrutura das expressões OCL pode influenciar no desempenho dos participantes em relação à correção do entendimento e ao tempo despendido para chegar a um entendimento sobre uma restrição OCL.

Expressões contendo *OCL smells* tiveram um menor índice de acerto em relação à correção do entendimento, e os participantes gastaram mais tempo para entendê-las. As avaliações subjetivas realizadas pelos participantes mostraram que as questões do tipo S foram julgadas como mais difíceis do que as questões do tipo R. Além disso, pudemos constatar uma correlação entre o julgamento subjetivo da dificuldade das questões e o desempenho dos participantes tanto do ponto de vista da correção do entendimento quanto do tempo necessário para resolvê-las.

Os resultados obtidos na avaliação subjetiva da qualidade das expressões presentes nas questões refletem, de certo modo, a inexperiência dos participantes com a OCL. Embora uma boa parte perceba que certas expressões tenham sido definidas de forma desnecessariamente complexa, poucos sugeriram caminhos concretos para simplificá-las. Entretanto, acreditamos que um motivo adicional para isto possa residir no fato do conhecimento sobre expressões potencialmente problemáticas e possíveis alternativas de estruturação, conforme proposto nesta tese, não estar disponível de forma explícita para os usuários de OCL.

Desta forma, o estudo apresenta indícios de que a técnica de reestruturação aplicada a restrições especificadas em OCL pode ser viável. Entretanto, repetições deste estudo, em outros contextos, além de outros estudos são necessários para reforçar as conclusões obtidas com este primeiro estudo, bem como para aprofundar as análises. A realização deste experimento de forma presencial com material impresso também se faz necessária, embora, segundo a nossa avaliação inicial, o fato dos dados terem sido

coletados em reuniões através da internet não exerceu influência significativa nos resultados.

Várias das reestruturações descritas nesta tese podem ser definidas formalmente e, portanto, são passíveis de automação. Mas esse não é o caso geral. O capítulo a seguir discute como as reestruturações podem ser automatizadas, e como uma infra-estrutura de apoio à elaboração e à execução de casos de teste de especificações OCL pode ser utilizada para apoiar a realização dessas reestruturações.

Capítulo 6

Reestruturações Automatizadas e Manuais

6.1 Introdução

Ainda que não contenham construções que possam comprometer o seu entendimento e a sua evolução, modelos com restrições especificadas em OCL precisam ser avaliados em relação à presença de problemas como, por exemplo, inconsistências, violações sintáticas, utilização incorreta de tipos, ou ainda, captura incorreta de fatos sobre o universo que está sendo representado. Além disso, as reestruturações aplicadas a um modelo, de forma manual, também podem introduzir problemas dessa natureza.

Uma das formas de reestruturar um modelo UML/OCL consiste em aplicar reestruturações automatizadas, isto é, especificadas e implementadas de forma a preservar a semântica do modelo. Entretanto, nem todas as reestruturações aplicáveis a um modelo podem ser totalmente automatizadas, especialmente aquelas que envolvam análises complexas de equivalência semântica entre operações (ROBERTS, 1999), ou que não possam ser precisamente especificadas, como é o caso de reestruturações como *Remover Redundância* e *Introduzir Polimorfismo*, por exemplo. Portanto, embora várias reestruturações possam ser automatizadas, outras deverão ser realizadas manualmente, seja pela inviabilidade técnica ou econômica de disponibilizá-las de forma automatizada, seja por elas ainda não estarem disponíveis de forma automatizada.

A realização de reestruturações tanto automatizadas como manuais é uma realidade também em ambientes de programação, que disponibilizam para seus usuários, de forma automatizada, apenas um subconjunto das reestruturações disponíveis em catálogos como (FOWLER, 1999). Nas reestruturações disponíveis de forma automatizada, a preservação da semântica do sistema deve ser garantida pelo ambiente que as implementa. Por outro lado, as reestruturações manuais são encaradas como operações que geram uma nova versão do sistema, onde a avaliação da preservação da semântica do sistema é apoiada pela execução automática de testes de regressão, que visam avaliar se a nova versão continua realizando as mesmas funções, e

gerando os mesmos resultados como em sua versão anterior (FOWLER, 1999), (SAFF e ERNST, 2004), (GEORGE e WILLIAMS, 2004).

O restante deste capítulo apresenta a abordagem proposta para apoiar a automação de reestruturações, bem como a verificação da preservação da semântica do modelo, e está estruturado da seguinte forma: a seção 6.2 apresenta uma proposta para a automação de reestruturações aplicadas a modelos com restrições expressas em OCL, que é baseada na definição de operações de transformação. As condições para a aplicação de uma reestruturação, bem como os efeitos esperados, são especificados em OCL. As ações de transformação são definidas em uma linguagem de ações, a OCL-AL (OCL Action Language), que estende a OCL de modo a permitir a elaboração de especificações executáveis de operações. A seção 6.3 descreve como a execução automatizada de testes de regressão pode ser utilizada tanto para apoiar a avaliação da semântica do modelo produzido, como para apoiar a avaliação da preservação da semântica do modelo após a realização de reestruturações. Finalmente, a seção 6.4 apresenta algumas considerações finais.

6.2 Reestruturações Automatizadas

Expressões OCL podem ser utilizadas tanto em modelos UML, como em metamodelos definidos a partir do MOF. De acordo com a arquitetura de metamodelos definida pelo OMG, um modelo UML (nível M1) é definido por instâncias de elementos do metamodelo da UML (nível M2). O metamodelo da UML, por sua vez, é definido por instâncias de elementos do MOF (nível M3). O metamodelo da OCL 2.0 define associações entre alguns dos seus elementos e elementos do metamodelo da UML. Desta forma, um modelo UML com restrições especificadas em OCL pode ser visto como uma coleção de instâncias e ligações entre instâncias de elementos definidos nos dois metamodelos (UML e OCL). O metamodelo da UML e as restrições para modelos bem formados, especificadas em OCL, podem ser vistos como uma coleção de instâncias e ligações entre instâncias de elementos definidos no MOF e no metamodelo da OCL.

A Figura 6.1 apresenta um exemplo de como uma expressão OCL é representada na forma de instâncias desses metamodelos. O diagrama de objetos desse exemplo corresponde à representação parcial da expressão *a < 20 implies (b <= 10 implies c >*

0), onde *a*, *b* e *c* são atributos definidos na classe *X* de um modelo. Essa expressão foi definida no contexto da classe *X*. *OperationCallExp, AttributeCallExp, IntegerLiteralExp, VariableExp* e *Variable* são classes definidas no metamodelo da OCL, enquanto que *Attribute, Operation* e *Class* são classes definidas no metamodelo da UML.

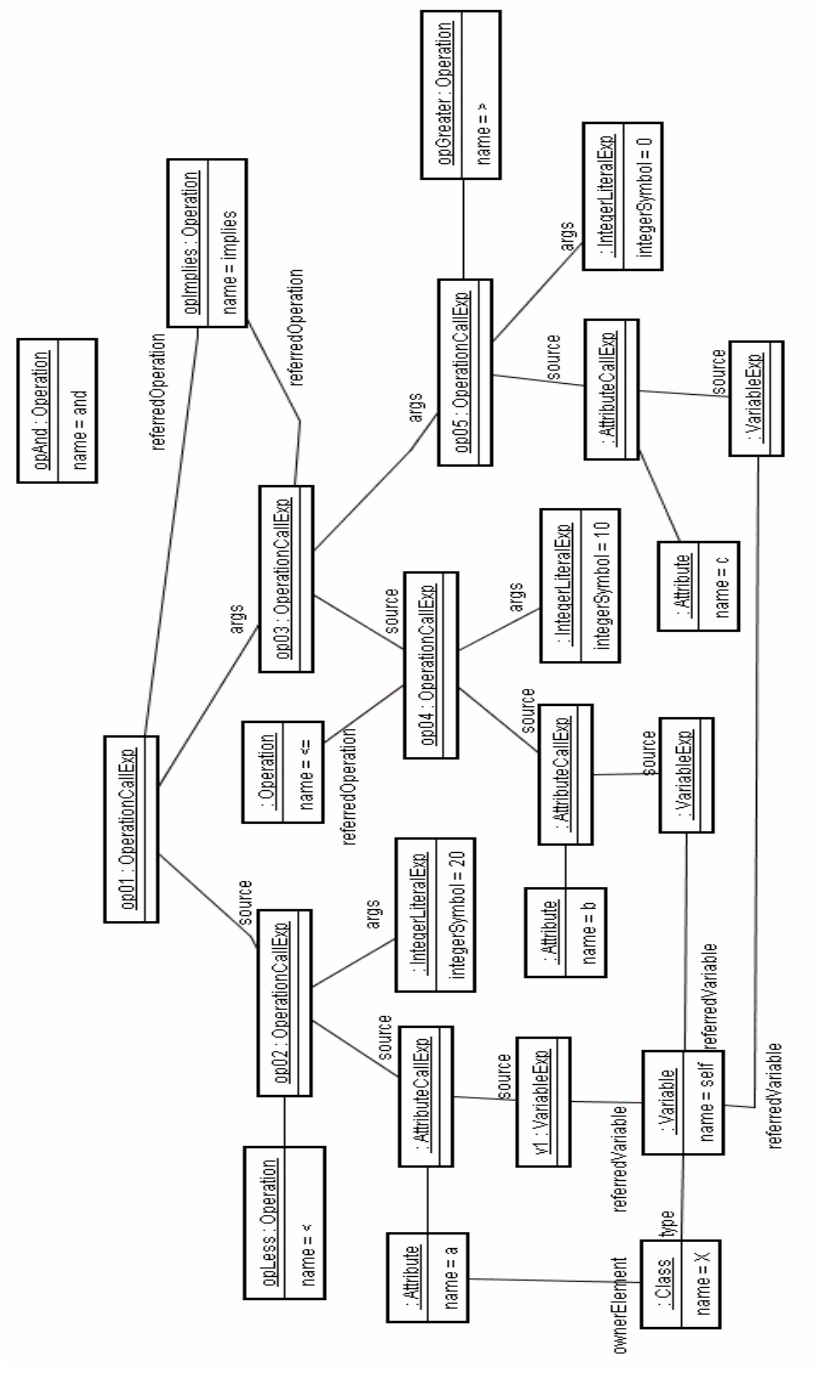


Figura 6.1 – Diagrama de objetos correspondente à expressão *a < 20 implies (b <= 10 implies c > 0*

Uma expressão do tipo *A implies B* é representada por uma instância da classe *OperationCallExp.* Toda instância de *OperationCallExp* está associada a três elementos:

- *referredOperation*: operação referenciada pela expresão. Essa operação deve estar definida em uma classe do modelo subjacente ou em um dos tipos definidos pela OCL.
- *source:* elemento que receberá a invocação da operação, e, que neste exemplo, corresponde à expressão *A;*
- *arguments:* expressões que correspondem aos argumentos da operação. A operação *implies,* por exemplo, espera apenas uma expressão como argumento. Na expressão *A implies B*, o argumento corresponde à expressão *B*.

As referências aos atributos *a*, *b* e *c* na expressão ilustrada pela Figura 6.1 são representadas como instâncias da classe *AttributeCallExp*. Neste exemplo, o objeto origem (*source)* associado a cada uma dessas instâncias de *AttributeCallExp* corresponde ao objeto referenciado pela variável *self,* que é utilizada de forma implícita na expressão.

A realização de uma reestruturação pode ser vista como uma das possíveis transformações aplicáveis a um modelo. De um modo geral, uma transformação pode ser definida através de um mapeamento de um modelo origem em um modelo alvo, ou então, como uma atualização do próprio modelo origem. Em um mapeamento, elementos do modelo origem são transformados em zero, um ou mais elementos do modelo alvo, sem que o modelo origem seja alterado. A geração de código Java a partir de um modelo UML, por exemplo, pode ser definida através de uma transformação baseada em mapeamento. Uma transformação baseada na atualização do próprio modelo origem consiste na adição, modificação ou eliminação de elementos em um modelo, ou seja, o modelo alvo é produzido por atualizações efetuadas no próprio modelo origem. Em geral, transformações de atualização operam sobre um pequeno subconjunto do modelo origem.

Uma reestruturação aplicada a um modelo com restrições em OCL pode ser definida, portanto, como uma transformação baseada na atualização do modelo, ou seja, como uma operação de transformação realizada sobre o conjunto de instâncias que representa o modelo. Dessa forma, o modelo reestruturado é obtido como resultado da

adição, modificação ou eliminação de objetos ou ligações entre objetos no modelo original.

O exemplo a seguir apresenta, informalmente, os efeitos produzidos por uma operação de reestruturação sobre a expressão OCL apresentada na Figura 6.1. Seja *seqImplies* uma seqüência de duas instâncias de *OperationCallExp* que referenciam a operação *implies*, correspondente à ocorrência do *OCL Smell Cadeia de Implicações* (seção 3.2). No exemplo da Figura 6.1, *seqImplies = Sequence{op01, op03}*. A reestruturação *Substituir Cadeia de Implicações por Uma Única Implicação* pode ser definida como uma operação de atualização que recebe *seqImplies* como parâmetro, e gera os seguintes efeitos na expressão alvo:

a) Para cada elemento de *seqImplies,* com exceção do último:

- Sua ligação com a instância de *Operation* referente à operação *implies* será eliminada. No exemplo, a ligação entre *op01* e *opImplies* será eliminada.
- Será criada uma ligação com a instância de *Operation* correspondente à operação *and* do classificador *Boolean*. No exemplo, será criada uma ligação entre *op01* e *opAnd.*
- Sua ligação com a instância de *OperationCallExp* subseqüente em *seqImplies* será eliminada. No exemplo, a ligação entre *op01* e *op3* será eliminada.
- Será criada uma ligação com a expressão origem (*source*) ligada à instância de *OperationCallExp* subseqüente em *seqImplies*. Essa expressão corresponderá à propriedade *args* de *OperationCallExp.* No exemplo, *op03* é a instância subseqüente de *op01*, e *op04* é a expressão fonte de *op03*. Portanto, será criada uma ligação entre *op01* e *op04,* onde op04 exercerá o papel de *args.*

b) Para cada elemento de *seqImplies,* com exceção do primeiro:

- Será eliminada a ligação com sua respectiva expressão fonte *(source)* existente no momento da invocação da operação de reestruturação. No exemplo, a ligação entre *op03* e *op04* será eliminada.

- Será criada uma ligação com a instância de *OperationCallExp* precedente em *seqImplies,* que corresponderá à propriedade *source.* No exemplo, será criada uma ligação entre *op03* e *op01*, onde *op01* exercerá o papel de *source.*

A Figura 6.2 apresenta o diagrama de objetos resultante da aplicação desta reestruturação sobre o exemplo apresentado na Figura 6.1. Esse diagrama corresponde à expressão reestruturada: (*a < 20 and b <= 10) implies c > 0.*

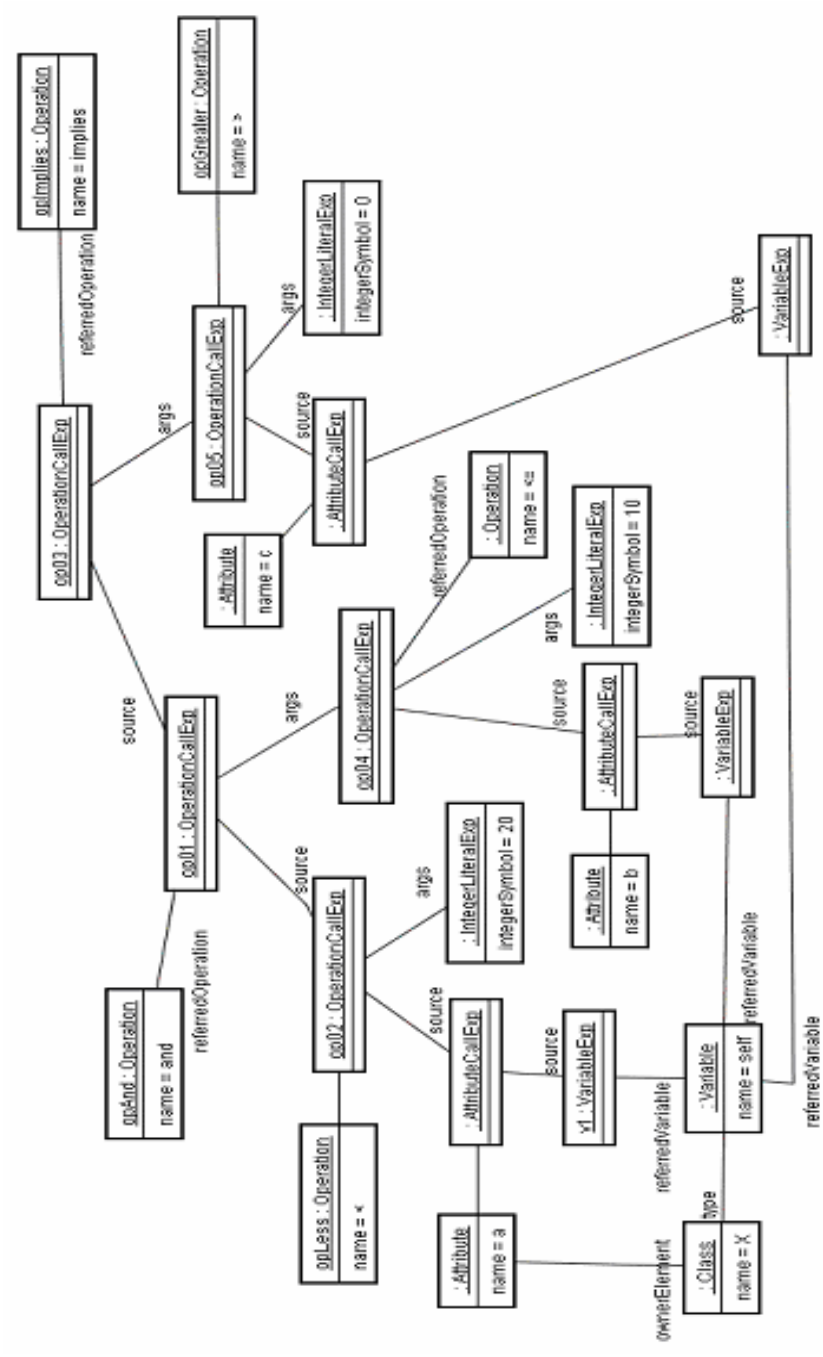


Figura 6.2 – Diagrama de objetos da expressão reestruturada (a < 20 and b <= 10) implies c > 0

```
 1 context ReplaceImpliesChainRefactoring::refactor(
 2            seqImplies : Sequence(OperationCallExp))
 3 def: allButLast : Sequence(OperationCallExp) =
 4      seqImplies->subsequence(1, seqImplies->size() – 1)
 5 def: allButFirst : Sequence(OperationCallExp) =
 6      seqImplies->subsequence(2, seqImplies->size())
 7 def: first : OperationCallExp = seqImplies->at(1)
 8 def: last : OperationCallExp = seqImplies->at(seqImplies->size())
 9
10 modifiable: links(seqImplies, arguments)
11 modifiable: links(seqImplies, source)
12 modifiable: links(seqImplies, referredOperation)
13
14 –- seqImplies deve conter pelo menos duas chamadas de operação
15 pre: seqImplies->size() >= 2
16 -- seqImplies deve conter apenas chamadas à operação implies
17 pre: seqImplies->forAll(opCallExp |
18      opCallExp.referredOperation = impliesOp)
19 -- a ordem dos elementos de seqImplies deve corresponder
20 -- ao encadeamento das chamadas à operação implies
21 pre: allButLast->forAll(opCallExp |
22      opCallExp.firstArgument = next(seqImplies, opCallExp))
23
24 –- chamadas passaram a referenciar a operação and
25 post:
26  allButLast->forAll(e | e.referredOperation = andOp)
27 -- com exceção da última, que continua referenciando implies
28 post:  last.referredOperation = impliesOp
29 –- argumento da operação and corresponde à antiga expressão fonte
30 -- do próximo elemento
31 post:  allButLast->forAll(e | e.firstArgument =
32                    next(seqImplies, e).source@pre)
33 –- com exceção do ultimo elemento, cujo argumento continua o mesmo
34 post: last.firstArgument = last.firstArgument@pre
35 –- expressão fonte passa a ser a expressão anterior na cadeia
36 post: allButFirst->forAll(e | e.source = prev(seqImplies, e))
37 –- com exceção do primeiro elemento
38 post: first.source = first.source@pre
```

Figura 6.3 – Contrato da reestruturação Substituir Cadeia de Implicações por uma Única Implicação

Os efeitos produzidos por uma operação de reestruturação, bem como as restrições que precisam ser satisfeitas antes da aplicação da reestruturação, podem ser especificados por pré e pós-condições. A Figura 6.3 apresenta a definição da reestruturação *Substituir Cadeia de Implicações por Uma Única Implicação*. Ela corresponde ao classificador *ReplaceImpliesChainRefactoring*, e a sua semântica é definida pela operação *refactor,* que recebe como parâmetro uma seqüência de chamadas à operação *implies*. As pré-condições são especificadas nas linhas 14-22, e estabelecem que o parâmetro *seqImplies* deve conter pelo menos duas chamadas, em seqüência, à operação implies. As pós-condições (linhas 24-38) definem os efeitos dessa reestruturação, descritos informalmente no exemplo anterior. A Figura 6.4 lista as definições auxiliares que são utilizadas nas pré e pós-condições dessa operação.

```
 1 context ReplaceImpliesChainRefactoring
 2 def: booleanType: Classifier =
 3     Classifier::allInstances()->select(name = 'Boolean')->any(true)
 4 def: impliesOp : Operation =
 5      booleanType.lookupOperation('implies', Sequence{booleanType})
 6 def: andOp : Operation =
 7      booleanType.lookupOperation('and', Sequence{booleanType})
 8
 9 def: next(seq : Sequence(OperationCallExp), e : OperationCallExp) :
10      OperationCallExp = seq->at(seq->indexOf(e) + 1)
11 def: prev(seq : Sequence(OperationCallExp), e : OperationCallExp) :
12      OperationCallExp = seq->at(seq->indexOf(e) - 1)
13
14 context OperationCallExp
15 def: isReferredOperation(type : Classifier, name : String):Boolean=
16      self.referredOperation.getOwner() = type and
17      self.referredOperation.getName() = name
18 def: firstArgument : OclExpression =
19      self.arguments->first()
```

Figura 6.4 – Definições auxiliares para a reestruturação Substituir Cadeia de Implicações por uma Única Implicação

A definição da reestruturação apresentada na Figura 6.3 utilizou duas extensões à linguagem OCL definidas nesta tese. A primeira corresponde à utilização da construção *def* para definir elementos que possam ser utilizados localmente nas restrições de uma operação. Na versão original da linguagem, a cláusula *def* só pode ser utilizada no

contexto de um classificador. A expressão associada à variável *allButLast,* definida nas linhas 3-4 da Figura 6.3, por exemplo, é utilizada nas linhas 21, 26 e 31.

A segunda extensão corresponde à construção *modifiable,* que estabelece restrições em relação ao escopo das modificações admissíveis em função da execução da operação. As restrições definidas nas linhas 10-12 da Figura 6.3 estabelecem que a operação *refactor* pode gerar modificações somente nas ligações entre os objetos pertencentes à *seqImplies* e os objetos designados pelos papéis *arguments, source* e *referredOperation.* O capítulo 7 apresenta uma descrição detalhada dessas extensões.

Embora a OCL possa ser utilizada para especificar os efeitos esperados por uma operação, como por exemplo, a reestruturação apresentada na Figura 6.3, as ações que devem ser executadas para gerar esses efeitos não podem ser especificadas em OCL. A partir da sua versão 1.5, a especificação da UML passou a definir uma semântica de ações, a UML Action Semantics (UML AS), que fornece uma forma padronizada e independente de plataforma para a especificação de ações e atividades em um modelo, tornando possível a elaboração de modelos executáveis.

Como a UML AS não define uma sintaxe concreta para a especificação das ações, definimos a OCL-AL (OCL Action Language), uma linguagem que estende a OCL de forma a permitir a especificação de ações que atuam sobre instâncias no nível M0 (instâncias de um modelo UML, por exemplo), bem como de instâncias no nível M1 (instâncias de um meta-modelo UML, por exemplo). Através da OCL-AL, é possível especificar ações associadas a operações tanto de modelos M1 como de modelos M2, tornando possível a realização de atividades como, por exemplo, a execução de operações de um modelo UML, assim como de operações de manipulação e transformação aplicáveis a esses modelos como, por exemplo, reestruturações e aplicação de padrões. Assim, especificações implícitas, produzidas com a OCL, podem ser combinadas com especificações explícitas produzidas com a OCL-AL. A linguagem OCL-AL é descrita detalhadamente no capítulo 7.

A Figura 6.5 apresenta um exemplo de especificação explícita de ações elaborada com a OCL-AL. Esse exemplo corresponde às ações que devem ser executadas na reestruturação *Substituir Cadeia de Implicações por Uma Única Implicação*. A especificação completa dessa reestruturação corresponde à combinação das linhas de 4-14 da Figura 6.5 (especificação explícita) com a especificação definida na Figura 6.3 (especificação implícita).

```
 1 context ReplaceImpliesChainRefactoring::refactor(
 2            seqImplies : Sequence(OperationCallExp))
 3
 4 actionBody:
 5 begin
 6     foreach opCallExp in allButLast do
 7     begin
 8         opCallExp.referredOperation := andOp;
 9         opCallExp.arguments :=
10                Sequence{next(seqImplies, opCallExp).source};
11     end;
12     foreach opCallExp in allButFirst do
13         opCallExp.source := prev(seqImplies, opCallExp);
14 end
```

Figura 6.5 – Exemplo das ações associadas à reestruturação Substituir Cadeia de Implicações por uma Única Implicação

As ações de uma operação são definidas através da construção *actionBody*. Essa construção pode ser definida somente para as operações que possam provocar modificações no espaço de objetos sobre o qual elas atuem. Uma construção *actionBody* está associada a uma ação ou a um bloco de ações delimitado por *begin-end*. As ações descritas nas linhas 8-10 da Figura 6.5 são efetuadas para cada objeto *objCallExp* presente na coleção *allButLast* definida na Figura 6.3. Essas ações eliminam as ligações existentes entre *opCallExp* e os objetos definidos pelos papéis *referredOperation* e *arguments*, e criam ligações entre *opCallExp* e os objetos definidos pelas respectivas expressões definidas à direita do operador de atribuição (:=). Esse exemplo mostra que as ações de modificação e as estruturas de repetição são especificadas através de construções específicas da OCL-AL (*foreach, :=)*, enquanto que as ações de leitura são definidas através de expressões OCL (*allButLast*, *Sequence{next(seqImplies, opCallExp).source},* etc).

6.3 Reestruturações Manuais e Testes de Regressão

Embora diversas reestruturações possam ser automatizadas através da abordagem descrita na seção anterior, existem situações onde as reestruturações têm que ser realizadas de forma manual, seja em função da impossibilidade de formalizá-las, seja pelo fato de elas ainda não estarem disponíveis de forma automatizada.

No caso das reestruturações manuais de um modelo, adaptamos a abordagem empregada na reestruturação manual de código, que consiste na realização de testes de regressão automatizados como uma das formas de avaliar a preservação da semântica após as modificações realizadas manualmente. Mesmo que todas as reestruturações possíveis estivessem disponíveis de forma automatizada em uma ferramenta, o desenvolvimento de casos de testes ainda seria útil como apoio à validação da semântica das especificações OCL (FOWLER, 1999).

A Figura 6.6 apresenta um extrato de um diagrama de classes que será utilizado nos exemplos apresentados nesta seção.

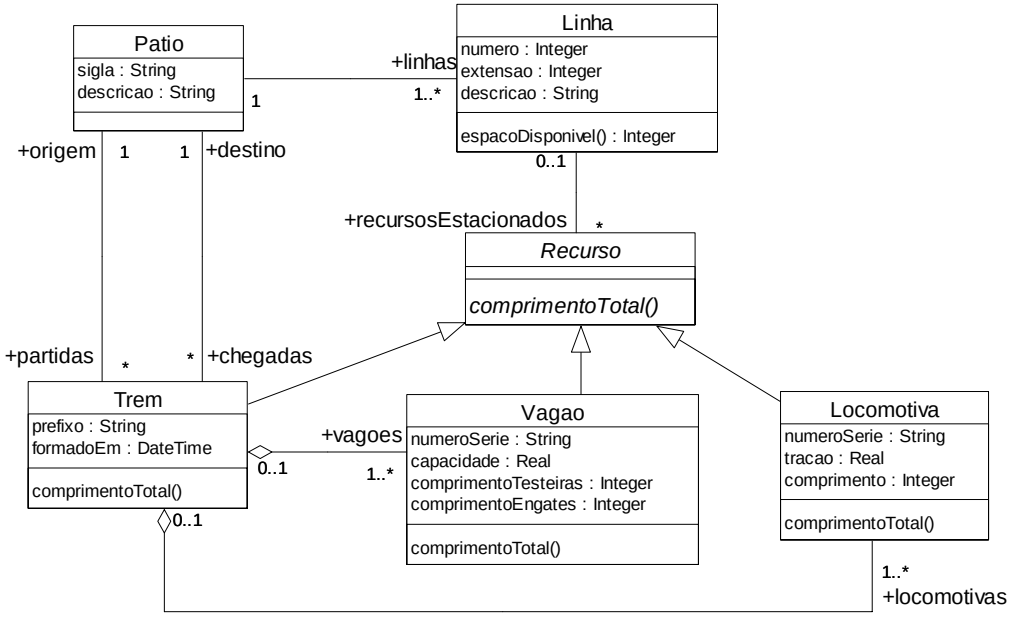


Figura 6.6 – Extrato de um diagrama de classes de um sistema de logística de transporte ferroviário

Esse diagrama foi extraído do modelo de um sistema para logística de transporte ferroviário. Um pátio é um local de referência na malha ferroviária onde são realizadas operações como carga e descarga de trens, formação de novos trens, entre outras. Um

pátio possui diversas linhas ferroviárias onde os recursos podem ser estacionados e manobrados. Um recurso pode ser uma locomotiva, um vagão, ou um trem. Um trem é formado por vagões e locomotivas, e tem origem em um pátio e destino em outro pátio da malha ferroviária.

6.3.1 Avaliação de expressões OCL e de especificações explícitas de operações que não modificam o estado do sistema

A definição de um caso de teste para especificações explícitas de operações de consulta, ou para expressões associadas a atributos derivados ou a atributos definidos através da construção *def,* é formada por três elementos:

- um espaço de objetos definindo uma configuração de objetos e ligações entre objetos.
- uma expressão OCL que será avaliada sobre o espaço de objetos definido no item anterior.
- o resultado esperado para a avaliação dessa expressão OCL.

A Figura 6.7 e a Figura 6.8 apresentam alguns exemplos de especificações explícitas para operações de consulta. Cada construção *body* define, explicitamente, o resultado da respectiva operação através de uma expressão OCL.

```
 1 -- comprimento total de um vagão é a soma do comprimento entre
 2 -- as testeiras e do comprimento dos engates localizado na suas
 3 -- extremidades
 4 context Vagao::comprimentoTotal() : Integer
 5 body: comprimentoTesteiras + comprimentoEngates
 6
 7 context Locomotiva::comprimentoTotal() : Integer
 8 body: comprimento
 9
10 -- o comprimento total de um trem é o somatório do comprimento
11 -- total de todos os seus vagões e suas locomotivas
12 context Trem::comprimentoTotal() : Integer
13 body: vagoes.comprimentoTotal()->sum() +
14       locomotivas.comprimentoTotal()->sum()
```

Figura 6.7 – Especificação explícita das operações comprimentoTotal dos Recursos

```
-- o espaço disponível em uma linha é dado pela extensão da linha
1 -- menos o somatório do comprimento total de todos os recursos
2 -- (trens, vagões e locomotivas) nela estacionados.
3 context Linha::espacoDisponivel() : Integer
4 body: extensao – recursosEstacionados.comprimentoTotal()->sum()
```

Figura 6.8 – Especificação explícita da operação espacoDisponivel da classe Linha

A Figura 6.9 apresenta um diagrama de objetos correspondente ao extrato de um espaço de objetos que pode ser utilizado na definição de alguns casos de teste para as operações listadas na Figura 6.7. Nesse exemplo, o pátio *BRM* (*patio1*) possui duas linhas. A linha 2 está vazia, enquanto que o trem NAG1020, que possui dois vagões e uma locomotiva, encontra-se estacionado na linha 1. Esse trem tem origem no pátio *VRR* e seu destino é o pátio *GBB*.

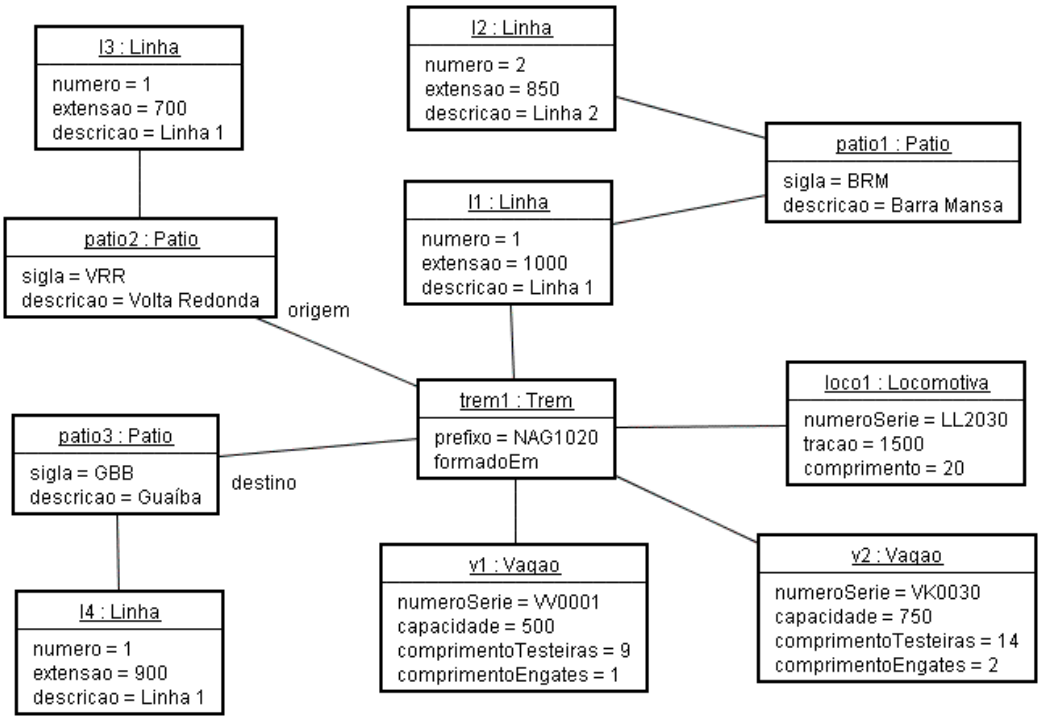


Figura 6.9 – Exemplo de um espaço de objetos correspondente ao sistema de logística de transporte ferroviário

A Figura 6.10 apresenta alguns casos de teste para as operações listadas na Figura 6.7. Cada caso de teste é definido por uma expressão OCL (*e1*) a ser avaliada, uma expressão OCL (*e2*) que corresponde ao valor esperado, e pelo espaço de objetos que será utilizado na avaliação das expressões *e1* e *e2*. Todos os casos de teste deste

exemplo estão associados ao espaço de objetos apresentado na Figura 6.9. Casos de teste para expressões associadas com atributos derivados, assim como as operações e atributos definidos através da construção *def,* podem ser definidos de forma análoga.

Expressão a ser avaliada	Valor Esperado
`v1.comprimentoTotal()`	10
`V2.comprimentoTotal()`	16
`loco1.comprimentoTotal()`	20
`trem1.comprimentoTotal()`	46
`l1.espacoDisponivel()`	954
`l2.espacoDisponivel()`	850

Figura 6.10 - Exemplos de casos de teste para uma operação de consulta

6.3.2 Avaliação de pré e pós-condições através de execução simulada

Esta seção descreve como as pré e pós-condições associadas a uma operação podem ser avaliadas através de execução simulada. A abordagem utilizada para a avaliação da especificação implícita de uma operação consiste em definir os efeitos gerados em um cenário específico de sua execução como se ela tivesse sido efetivamente executada. Um cenário de execução simulada de uma operação corresponde à definição dos seguintes elementos:

- o espaço de objetos (*e1*) correspondente ao estado do sistema no instante da invocação da operação;
- a sentença correspondente à invocação da operação, especificando o objeto que vai receber a invocação e os argumentos;
- o espaço de objetos (*e2*) correspondente ao estado do sistema obtido após a execução da operação neste cenário de invocação. No caso de operações de consulta, *e1* e *e2* devem ser o mesmo espaço de objetos.
- o resultado retornado pela operação, caso a operação defina um resultado de retorno.

A definição desses elementos corresponde ao conceito de *filmstrip* definido em (D'SOUZA e WILLS, 1998). A Figura 6.11 apresenta a especificação de uma operação do sistema de logística de transporte ferroviário que pode provocar modificações no estado do sistema. A operação *moverRecurso* tem o propósito de mover um recurso para

uma linha destino. O recurso a ser movimentado e a linha de destino são parâmetros dessa operação. Além das pré e pós-condições que são definidas na OCL para especificar as condições e os efeitos esperados pela execução de uma operação, esse exemplo contém ainda uma construção *modifiable* (linhas 3-5) que define o escopo das possíveis modificações admitidas para essa operação.

```
 1 context SistemaLogistica::moverRecurso(
 2         recurso : Recurso, linhaDestino : Linha)
 3    -- a operação somente pode gerar modificações na ligação entre
 4    -- o recurso e a sua linha.
 5    modifiable: links(recurso, linha)
 6    pre recursoDefinido:
 7        not recurso.oclIsUndefined()
 8    pre linhaDefinida:
 9        not linhaDestino.oclIsUndefined()
10    pre recursoMovimentavel:
11    --  recurso tem que estar estacionado em alguma linha
12        recurso.linha->notEmpty()
13    pre movimentacaoParaOutraLinha:
14    --  recurso tem que ser movido para uma linha diferente
15    --  da qual o recurso esteja estacionado
16        linhaDestino <> recurso.linha
17    pre movimentacaoNoMesmoPatio:
18    --  linha origem tem que ser do mesmo patio da linha destino
19        linhaDestino.patio = recurso.linha.patio
20    pre temEspacoNaLinha:
21    --  existe espaço disponível na linha destino para o recurso
22        linhaDestino.temEspacoPara(recurso)

23   post recursoNaLinhaDestino:
24   --   recurso passou a estar estacionado na linha destino
25        linhaDestino.recursosEstacionados->includes(recurso)
26   post recursoSaiuDaLinhaOrigem:
27   --   recurso não está mais estacionado na linha origem
28        recurso.linha@pre.recursosEstacionados->excludes(recurso)
```

Figura 6.11 – Exemplo de especificação de pré e pós-condições de uma operação que gera modificações no estado do sistema

Um cenário de execução simulada da operação *moverRecurso* pode ser definido, por exemplo, pelos seguintes elementos:

a) Estado do sistema antes da execução da operação: representado, neste exemplo, pelo espaço de objetos apresentado na Figura 6.9.

b) Sentença correspondente à invocação da operação *moverRecurso: SistemaLogistica::moverRecurso(trem1, l2)*. Essa invocação corresponde à solicitação para que o trem *trem1* seja movimentado da linha *l1* para a linha *l2*

c) Estado do sistema após a execução da operação: representado, neste exemplo, pelo espaço de objetos apresentado na Figura 6.12. As diferenças entre esse espaço de objetos e aquele da Figura 6.9 são: a presença da ligação entre o trem *trem1* e a linha *l2*, e a eliminação da ligação entre *trem1* e a linha *l1*.

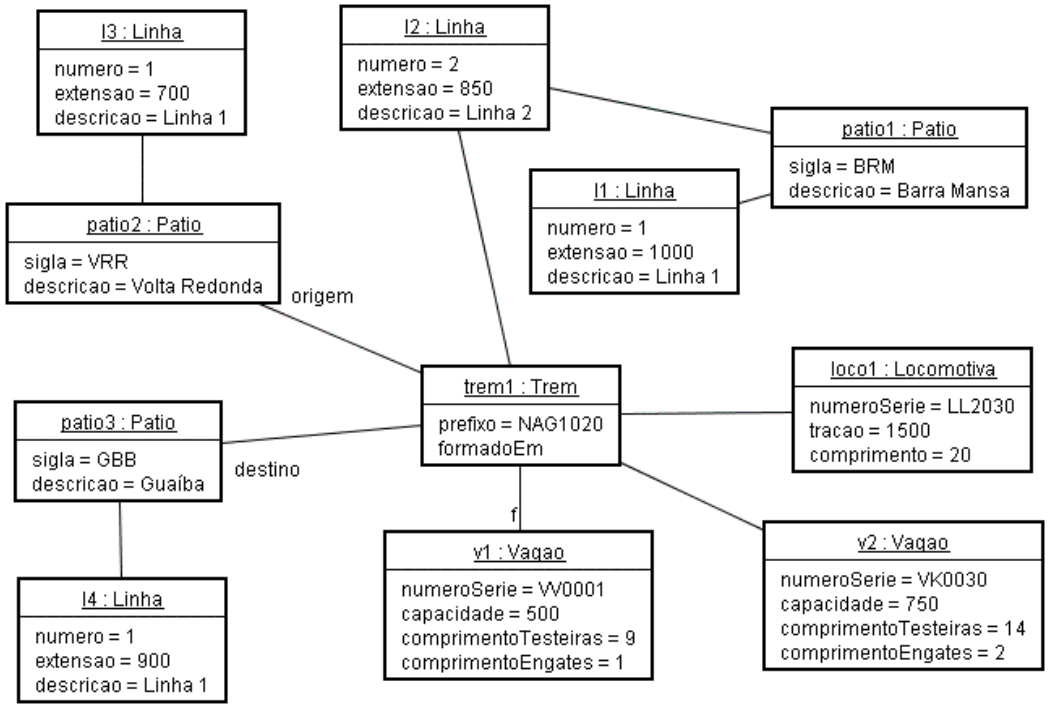


Figura 6.12 – Espaço de objetos após a execução da operação moverRecurso

Um caso de teste através da execução simulada de uma operação consiste na definição do cenário de execução simulada, da forma descrita anteriormente, e dos

resultados esperados da avaliação das restrições nesse cenário. Esses resultados são definidos pelos seguintes elementos:

- Resultado esperado da avaliação das pré-condições da operação, que pode assumir dois valores: *OK* ou *Falha*. O valor *OK* define que a avaliação de todas as pré-condições da operação deverá resultar no valor verdadeiro. O valor *Falha* indica que a avaliação de alguma das pré-condições da operação deverá resultar no valor falso ou indefinido. A avaliação das pré-condições é realizada sobre o espaço de objetos *e1* e os parâmetros recebidos pela operação.

- Resultado esperado da avaliação das pós-condições da operação que, de forma análoga à avaliação das pré-condições, pode assumir dois valores: *OK* ou *Falha*. A avaliação das pós-condições é realizada sobre os dois espaços de objetos (*e1* e *e2*) definidos no cenário de execução simulada da operação. Além disso, as modificações produzidas em *e1*, e que resultaram em *e2*, são avaliadas em relação às modificações admissíveis pela operação, especificadas através da construção *modifiable*.

- Resultado de retorno esperado, que corresponde a uma expressão OCL cujo valor será comparado com o resultado definido como retorno da operação. Essa expressão só precisa ser definida para operações que gerem um resultado de retorno.

A Figura 6.13 apresenta um exemplo de caso de teste para a operação *moverRecurso,* avaliado através de execução simulada.

Estado anterior do sistema (*e1*)	espaço de objetos apresentado na Figura 6.9.
Invocação	*SistemaLogistica::moverRecurso(trem1, l2)*
Estado posterior do sistema (*e2*)	espaço de objetos apresentado na Figura 6.12
Valor retornado pela operação	Não se aplica
Resultado esperado para a avaliação das pré-condições	OK
Resultado esperado para a avaliação das pós-condições	OK
Valor esperado para o retorno da operação	Não se aplica

Figura 6.13 – Exemplo de um cenário de teste através de execução simulada

6.3.3 Combinação de especificações implícita e explícita

Se uma operação possuir tanto uma especificação implícita (pré e pós-condições) como uma especificação explícita, as avaliações do resultado de retorno e das suas pós-condições podem ser realizadas sobre os efeitos produzidos pela execução direta das ações, no caso de operações modificadoras, ou da avaliação da expressão OCL, no caso de operações de consulta.

A especificação explícita de uma operação de consulta é definida por uma expressão OCL associada à operação através da construção *body*. Nesse caso, o espaço de objetos utilizado para a avaliação das pós-condições (*e2*) é o próprio espaço *e1*, e o resultado de retorno da operação é obtido a partir da avaliação da expressão OCL sobre o espaço *e1*.

No caso de uma operação modificadora, o espaço de objetos utilizado para a avaliação das pós-condições é gerado pela execução de sua especificação explícita que, neste caso, é definida por ações especificadas em OCL-AL e associadas à operação através da construção *actionBody*.

A Figura 6.14 apresenta a especificação explícita da operação *moverRecurso*. A ação definida para esta operação (linha 4) resulta na criação de uma ligação entre os objetos *recurso* e *linhaDestino* e na eliminação da ligação entre o objeto recurso e a linha à qual o recurso estava associado no momento da invocação da operação.

```
1 context SistemaLogistica::moverRecurso(
2         recurso : Recurso, linhaDestino : Linha)
3 actionBody:
4         recurso.linha := linhaDestino;
```

Figura 6.14 – Exemplo de especificação explícita de uma operação

Os casos de teste para operações que combinem especificação explícita com especificação implícita (pré e pós-condições) são definidos da mesma forma descrita na seção 6.3.2. A diferença é que as pós-condições são avaliadas sobre um espaço de objetos gerado a partir da execução das ações que compõem a especificação explícita da operação, ao invés de um espaço de objetos previamente definido *(e2),* como é o caso do cenário de execução simulada.

6.4 Considerações Finais

Este capítulo apresentou a abordagem proposta para a formalização e a automação de reestruturações, bem como para a verificação da preservação da semântica do modelo após a aplicação de reestruturações.

Uma reestruturação automatizada é especificada em uma classe que define uma operação de transformação (*refactor*). As condições para a aplicação de uma reestruturação são especificadas, em OCL, pelas pré-condições desta operação, enquanto que os efeitos esperados pela sua aplicação, são especificados pelas pós-condições desta operação. As ações de transformação são definidas em OCL-AL, uma linguagem que estende a OCL de modo a permitir a elaboração de especificações executáveis de operações.

A estratégia proposta para a formalização e a automação das reestruturações aplicáveis sobre um modelo com restrições em OCL é similar às abordagens para a reestruturação de modelos descritas em (SUNYÉ et al., 2001), que utiliza pré e pós-condições para especificar reestruturações aplicáveis a diagramas de estados, e (GORP et al., 2003), que também é baseada na utilização de pré e pós-condições para a especificação de reestruturações aplicáveis a diagramas de classes. A definição de reestruturações como operações de transformação especificadas através de pré e pós-condições é também utilizada na abordagem de reestruturação de código Smalltalk descrita em (ROBERTS, 1999). A essa especificação contratual de uma reestruturação, adicionamos a operacionalização da transformação na forma de ações especificadas em OCL-AL, reutilizando os mesmos elementos que podem ser empregados para a elaboração de modelos executáveis.

Uma questão relevante no contexto de transformações aplicadas a modelos é garantir a preservação da semântica do modelo após uma reestruturação, especialmente se ela for realizada de forma manual. A estratégia proposta nesta tese consiste em empregar testes de regressão executados de forma automatizada, reutilizando os casos de teste definidos para apoiar a avaliação da correção do modelo. Uma vez que os testes não provam que um modelo está semanticamente correto, nem que a versão reestruturada de um modelo é semanticamente equivalente à sua versão anterior, eles devem ser combinados com outras estratégias como provas formais e revisões, por

exemplo. Como qualquer estratégia baseada em análise dinâmica, o nível de confiança obtido está diretamente relacionado com a qualidade dos casos de teste utilizados.

Cabe ressaltar, entretanto, que tanto a elaboração de técnicas para a definição dos casos de teste para especificações elaboradas em OCL, quanto a prova formal da preservação da semântica para cada uma das reestruturações descritas nos capítulos 3 e 4 não foram objetivos desta tese.

Capítulo 7

Extensões à OCL

7.1 Introdução

Um requisito essencial para a utilização da abordagem proposta no capítulo 6 é a existência de uma semântica bem definida para a OCL e para os elementos que formam o modelo. Embora exista uma definição formal para a OCL e para um subconjunto do modelo de classes, a OCL apresenta várias deficiências com relação à sua semântica, que se constituiram em obstáculos para a utilização prática da abordagem proposta. Além disso, conforme descrito no capítulo anterior, a automação das reestruturações é baseada na execução de um conjunto de ações associadas a operações de transformação do modelo. Essas ações não podem ser especificadas em OCL, uma vez que expressões OCL não são capazes de modificar o espaço de objetos onde elas são avaliadas. Portanto, a abordagem proposta para o apoio automatizado às reestruturações demandou a elaboração do conjunto de extensões à OCL descrito neste capítulo.

O restante deste capítulo está organizado da seguinte forma: a seção 7.2 apresenta uma extensão à OCL que permite a definição de variáveis locais a uma operação. Isto permite a reutilização de expressões na especificação de pré e pós-condições de operações de um modelo ou metamodelo como, por exemplo, nas operações de reestruturação do modelo. A seção 7.3 descreve uma extensão que possibilita a definição precisa do escopo das modificações permitidas por uma operação do modelo. Esta extensão possibilita um maior nível de precisão na especificação dos contratos das operações de reestruturação, assim como na avaliação dos casos de teste das operações do modelo. A seção 7.4 apresenta algumas extensões e correções efetuadas na definição de algumas operações da biblioteca padrão da OCL. Essa seção propõe correções na definição de algumas operações fundamentais para a implementação de várias reestruturações como, por exemplo, *select, exists* e *one.* A seção 7.5 descreve algumas modificações propostas à OCL referentes à definição de expressões de inicialização e derivação de valores de atributos, uma vez que a definição original restringia a realização de algumas reestruturações aplicáveis a hierarquias de

classes. A seção 7.6 apresenta a linguagem OCL-AL (OCL Action Language), que estende a OCL de modo a permitir a elaboração de especificações executáveis de operações, possibilitando a operacionalização das ações que realizam as reestruturações. Finalmente, a seção 7.7 apresenta as considerações finais deste capítulo.

7.2 Definições locais a uma operação

A OCL oferece dois mecanismos para evitar a ocorrência de duplicação de expressões. O primeiro mecanismo é a construção *let*, que consiste em definir uma variável que pode ser utilizada apenas no contexto da expressão subseqüente à palavra reservada *in*, como no exemplo da Figura 7.1.

```
context PessoaFisica
inv:
  let totalRecebimentos : Integer = self.recebimentos.valor->sum() in
       if  totalRecebimentos < 500
            then totalRecebimentos - 20
            else totalRecebimentos * taxaDesconto
       endif
```

Figura 7.1 – Exemplo de definição de uma variável em OCL

Se uma expressão tiver que ser utilizada em diferentes restrições, a duplicação pode ser evitada através da definição de um atributo ou operação auxiliar em uma classe do modelo. Isto pode ser feito através da construção *def,* como ilustrado no exemplo da Figura 7.2, onde o atributo *totalRecebimentos,* definido na classe *PessoaFisica*, foi utilizado em três restrições associadas a essa classe.

```
context PessoaFisica
def: totalRecebimentos : Integer  = self.recebimentos.valor->sum()
inv: if  totalRecebimentos < 500
         then totalRecebimentos - 20
         else totalRecebimentos * taxaDesconto
     endif
inv: totalRecebimentos >= 1000 implies taxaDesconto = 0.15
inv: totalRecebimentos < 1000 implies taxaDesconto = 0.10
```

Figura 7.2 – Exemplo de definição de um atributo auxiliar em OCL

Apesar das construções *let* e *def* evitarem a presença de várias situações de redundância em uma especificação OCL, existem algumas limitações cujos reflexos aparecem na especificação das pré e pós-condições de operações.

A principal limitação da construção *let* é que a variável por ela definida pode ser utilizada apenas na expressão *in* subseqüente. Isso significa que uma variável definida por uma expressão *let* não pode ser utilizada em mais de uma pré ou pós-condição de uma operação.

A cláusula *def,* por outro lado, resulta na adição de novos elementos (atributos ou operações) a uma classe do modelo, e sua principal limitação é não poder ser utilizada para evitar redundâncias em pós-condições que envolvam a operação *oclIsNew()* ou o operador *@pre*. Na pós-condição de uma operação *oper*, a expressão *obj.oclIsNew()* retorna o valor verdadeiro, caso *obj* tenha sido criado pela execução da operação *oper*. O operador *@pre* é utilizado para obter o valor de uma propriedade ou o resultado de uma operação de consulta aplicada a um objeto *obj*, considerando o estado desse objeto antes da execução da operação *oper*.

Uma vez que essas limitações dificultam a definição de pós-condições atômicas sem que ocorra duplicação de expressões, estendemos a OCL de modo a permitir a utilização de uma definição em mais de uma restrição (pré ou pós-condição) associada a uma operação. Essa extensão consiste na utilização da construção *def* no contexto de uma operação, que define uma variável que pode ser utilizada em qualquer restrição associada a essa operação. A sintaxe concreta para a definição de uma variável associada a uma operação é a mesma utilizada para definir um atributo auxiliar em uma classe, com exceção do contexto, que passa a ser uma operação.

Sejam dois espaços de objetos $\sigma(M)$ e $\sigma'(M)$ que correspondem, respectivamente, aos instantes da invocação e do término de execução de uma operação *oper*. As precondições da operação *oper* são avaliadas sobre o espaço $\sigma(M)$, enquanto que as suas pós-condições são avaliadas sobre $\sigma(M)$ e $\sigma'(M)$. As variáveis definidas nas cláusulas *def* associadas a uma operação *oper* podem ser utilizadas tanto nas pré-condições como nas pós-condições de *oper*. Entretanto, uma variável cuja expressão de inicialização associada contenha uma chamada para a operação *oclIsNew()* ou uma referência para o estado anterior (operador @pre), ou utilize, direta ou indiretamente, uma variável que tenha essas características, não pode ser utilizada nas pré-condições da operação. Na avaliação de uma pré-condição, o valor de uma variável é definido pela avaliação de sua

expressão sobre *σ(M)*. Na avaliação de uma pós-condição, o valor de uma variável é definido pela avaliação de sua expressão sobre *σ(M)* e *σ'(M)*.

A Figura 7.3 apresenta um diagrama com a definição de algumas classes de um sistema de vendas de produtos, enquanto que a Figura 7.4 apresenta um exemplo onde duas variáveis foram definidas e utilizadas em diferentes restrições da operação *registrarVenda.*

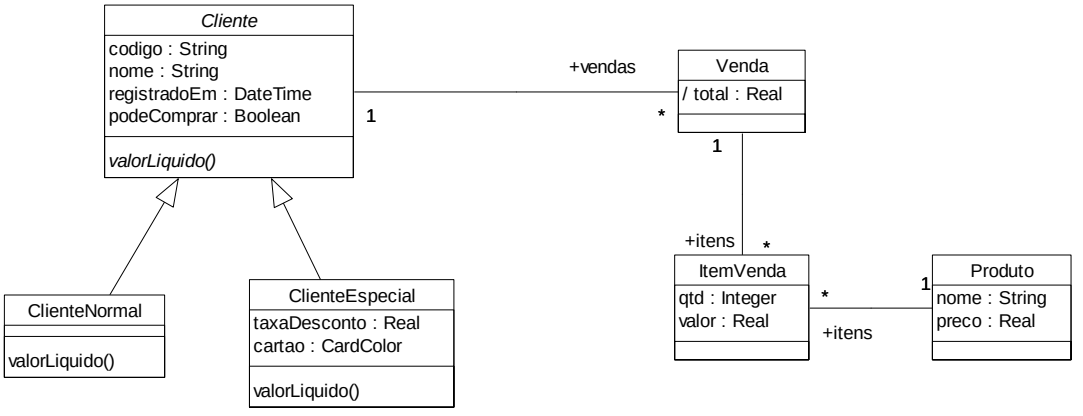


Figura 7.3 – Classes de um sistema de vendas de produtos

No exemplo da Figura 7.4, a variável *cliente* pode ser utilizada em todas as restrições da operação *registrarVenda*, sendo que o seu valor nas pré-condições é avaliado sobre o espaço *σ(M)* correspondente ao instante da invocação da operação, enquanto que o seu valor nas pós-condições é avaliado sobre o espaço *σ'(M)* correspondente ao instante em que a execução da operação foi concluída. Neste mesmo exemplo, a variável *novaVenda* pode ser utilizada somente nas pós-condições da operação, uma vez que a sua definição utiliza uma chamada à operação *oclIsNew*, cuja avaliação necessita dos espaços *σ(M)* e *σ'(M)*.

```
 1 context SistemaVendas::registrarVenda(
 2   pCodigo : String,
 3   pItens : Bag(Tuple(qtd : Integer, produto : Produto)))
 4 -- cliente – instância correspondente ao parâmetro pClientCode
 5 def: cliente : Cliente = Cliente::allInstances()->
 6                          select(c | c.codigo = pCodigo)->any(true)
 7 -- novaVenda – venda criada pela operação
 8 def: novaVenda : Venda = Venda::allInstances()->
 9                          select(v | v.oclIsNew())->any(true)
10
11 -- existe um cliente correspondente a pClienteCode
12  pre: not cliente.oclIsUndefined()
13  pre: cliente.podeComprar
14
15 -- apenas uma nova venda foi criada
16 post: Venda::allInstances() =
17       Venda::allInstances()@pre->including(novaVenda)
18 –- nova venda foi associada ao cliente
19 post: cliente.vendas = cliente.vendas@pre->including(novaVenda)
20 -- foram criados itens apenas para a nova venda
21 post: ItemVenda::allInstances() =
22       ItemVenda::allInstances()@pre + novaVenda.itens
23 post: novaVenda.itens->forAll(item | item.oclIsNew())
24 –- Para cada item solicitado foi criado um novo ItemVenda.
25 post: pItens = novaVenda.itens->collect(item |
26                Tuple {qtd = item.qtd, produto = item.produto} )
```

Figura 7.4 – Especificação de uma operação do sistema de vendas de produtos

7.3 Especificação do escopo das modificações admissíveis para uma operação

A OCL é uma linguagem de especificação do tipo restritiva, ou seja, as pós-condições de uma operação são expressas na forma de uma expressão booleana que restringe os valores permitidos para os elementos que compõem o estado do sistema após a execução da operação. Uma deficiência que a OCL apresenta consiste na dificuldade de se estabelecer as partes do sistema que não podem ser modificadas em função da execução de uma operação, problema conhecido como *"frame problem"* (MCCARTHY e HAYES, 1969). As operações definidas nas classes de um modelo

orientado a objetos podem ser divididas em duas grandes categorias: operações de consulta e operações modificadoras.

As operações de consulta não podem produzir modificações no estado do sistema. Dessa forma, se $\sigma(M)$ corresponder ao estado de um sistema representado pelo modelo M, e $\sigma'(M)$ corresponder ao estado do sistema obtido imediatamente após a execução de uma operação de consulta op_c, a propriedade $\sigma(M) = \sigma'(M)$ deverá sempre ser observada. Em OCL, o resultado esperado de operações de consulta pode ser especificado de forma explícita ou implícita.

A Figura 7.5 apresenta um diagrama de classes bastante simplificado de um sistema de gestão de contas bancárias, e um exemplo de especificação explícita de uma operação de consulta em OCL.

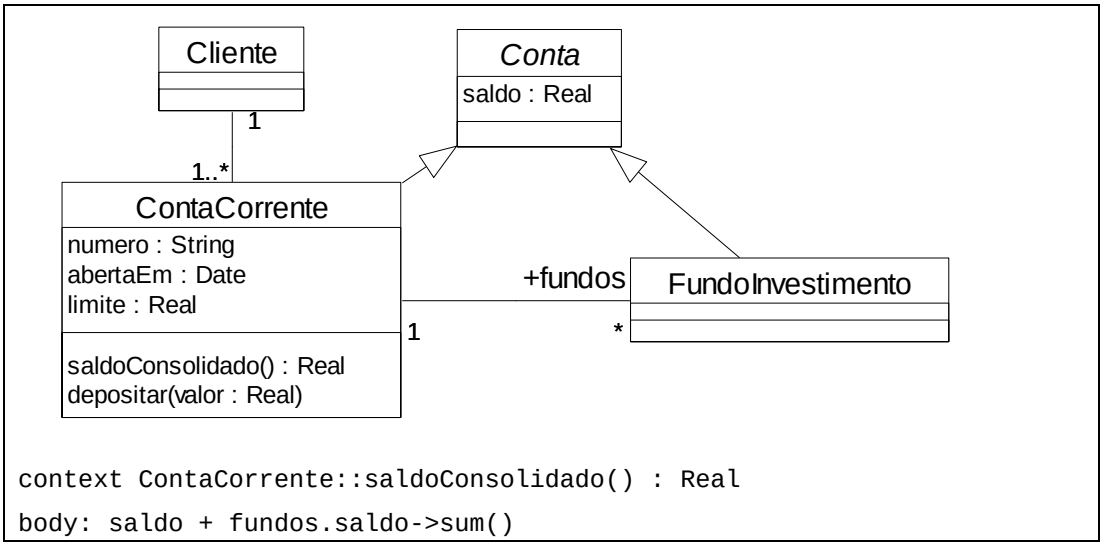


Figura 7.5 – Exemplo de especificação explícita de uma operação de consulta

Conforme as definições do diagrama, um cliente possui uma ou mais contas correntes, sendo que a cada conta corrente podem estar vinculadas várias contas de fundos de investimento. A operação *saldoConsolidado* fornece o saldo total de uma conta, incluindo o saldo dos fundos de investimento a ela vinculados. Nesse exemplo, a expressão *saldo + fundos.saldo->sum()* define explicitamente o valor esperado para o resultado dessa operação.

Uma operação de consulta pode ser especificada de forma implícita através de pré e pós-condições. Dois exemplos de especificações implícitas para a operação *saldoConsolidado* são apresentados na Figura 7.6 e na Figura 7.7. A pós-condição da

Figura 7.6 estabelece que o resultado da operação *saldoConsolidado* deve ser igual à soma dos valores da propriedade *saldo* da conta corrente e de seus respectivos fundos de investimento após a sua execução, enquanto que a pós-condição da Figura 7.7 estabelece apenas que o resultado deve ser maior que o *saldo* da conta corrente após a execução da operação.

```
context ContaCorrente::saldoConsolidado() : Real
post: result = saldo + fundos.saldo->sum()
```

Figura 7.6 – Exemplo de especificação implícita de uma operação de consulta

```
context ContaCorrente::saldoConsolidado() : Real
post: result >= saldo
```

Figura 7.7 – Exemplo de especificação implícita de uma operação de consulta

A definição da semântica de avaliação de pré-condições e pós-condições presente na especificação atual da OCL não faz distinção entre operações de consulta e operações modificadoras. Dessa forma, a restrição que estabelece que uma operação de consulta não deva produzir modificações no estado do sistema não é considerada na avaliação das suas pós-condições. Isso significa que implementações de uma operação de consulta que eventualmente modifiquem o estado do sistema seriam consideradas corretas, caso a avaliação das suas pós-condições fosse positiva.

Problemas semelhantes ocorrem com a especificação de operações modificadoras. Em OCL, operações modificadoras somente podem ser especificadas de forma implícita, ou seja, com pré e pós-condições. Embora modificações no espaço de objetos sejam permitidas como resultado da execução de uma operação, não é possível definir, de forma sucinta, o escopo permitido para essas modificações, ou seja, quais partes do espaço de objetos devem permanecer inalteradas. A Figura 7.8 apresenta a pós-condição especificada para a operação *depositar* definida na classe *ContaCorrente*, que estabelece que o *saldo* após a execução da operação deverá ser a soma do *saldo* no momento da invocação da operação e do valor recebido como parâmetro.

```
context ContaCorrente::depositar (valor : Real)
post: saldo = saldo@pre + valor
```

Figura 7.8 – Exemplo de especificação da operação *depositar* da classe *ContaCorrente*

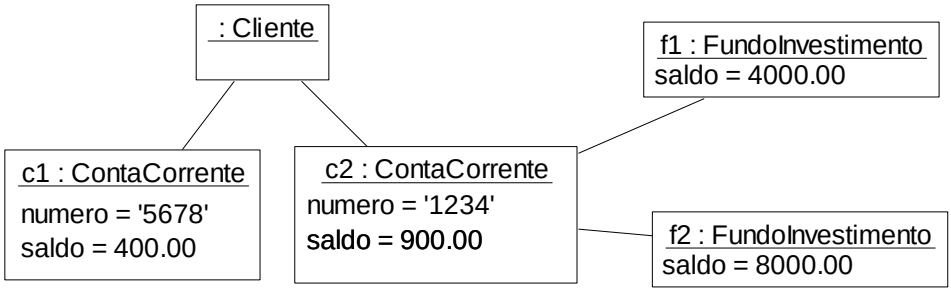


Figura 7.9– Espaço de objetos antes da execução de uma operação

Seja o cenário no qual a operação *depositar(200.00)* é solicitada ao objeto *c2* da classe *ContaCorrente* definido no espaço de objetos ilustrado na Figura 7.9, considerando três possíveis implementações para essa operação:

a) Implementação *I4*: resulta no universo de objetos ilustrado pela Figura 7.10, onde o atributo saldo do objeto *c2* passa a ser de 1100.00.

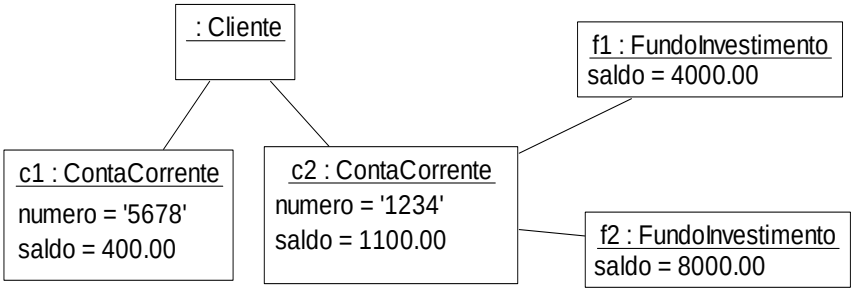


Figura 7.10 - Espaço de objetos obtido pela implementação *I4* da operação depositar

b) Implementação *I5:* resulta no universo de objetos ilustrado pela Figura 7.11, onde o atributo *saldo* do objeto *c2* passa a ser de 1100.00, e o atributo *saldo* do objeto *c1* passa a ser de 600.00.

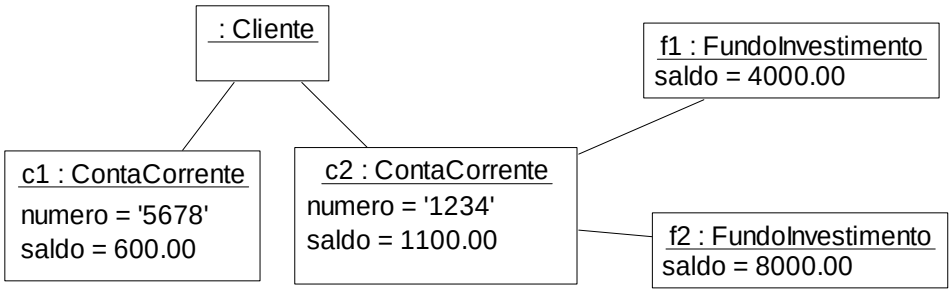


Figura 7.11 - Espaço de objetos obtido pela implementação *I5* da operação depositar

c) Implementação *I6* que resulta no universo de objetos ilustrado pela Figura 7.12, onde o atributo *saldo* do objeto *c2* passa a ser de 1100.00, e um objeto da classe *ContaCorrente* (*c3*) foi criado e associado ao objeto da classe *Cliente*.

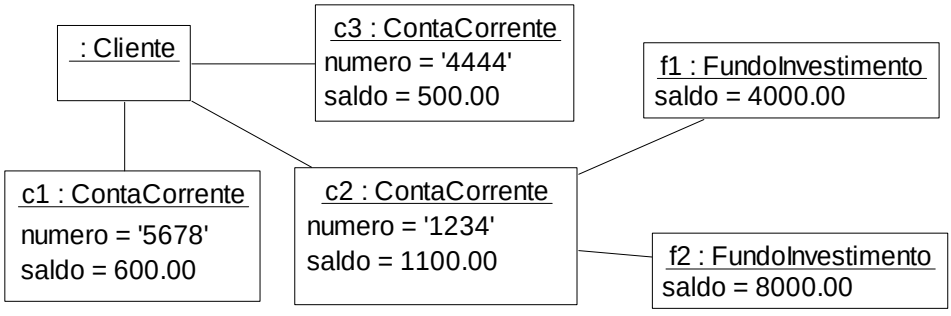


Figura 7.12 - Espaço de objetos obtido pela implementação *I6* da operação depositar

Embora o comportamento esperado nesse cenário fosse aquele gerado pela implementação *I4*, todas as implementações descritas (*I4, I5, I6*) seriam consideradas corretas de acordo com a especificação da operação *depositar*, pois a pós-condição "*post: saldo = saldo@pre + valor*" seria avaliada como verdadeira nas três implementações.

Uma forma de restringir o escopo de modificações admissíveis em uma operação é através da adição de expressões axiomáticas na forma de pós-condições da operação, explicitando as partes do sistema que devam permanecer inalteradas. A Figura 7.13 apresenta uma nova versão da especificação da operação *depositar* da classe *ContaCorrente* com algumas dessas expressões que visam definir o que deve permanecer inalterado (linhas 3-11).

Este exemplo demonstra que o número de expressões necessárias para restringir o escopo de modificações admissíveis por uma operação dificulta sobremaneira a utilização dessa estratégia, mesmo para modelos e operações muito simples. Um outro problema associado a essa estratégia está relacionado à evolução do modelo. A adição de uma classe (*A*, por exemplo) ao modelo implicaria em mudar as especificações de praticamente todas as operações, pois teríamos que acrescentar expressões do tipo *A::allInstances() = A::allInstances()@pre, A::allInstances()->forAll(a | a.attrib1 = a.attrib1@pre and a.atrib2 = a.atrib2@pre ...),* entre outras.

```
 1  context ContaCorrente::depositar(valor : Real)
 2  post: saldo = saldo@pre + valor
 3  post: numero = numero@pre
 4  post: limite = limite@pre
 5  post: abertaEm = abertaEm@pre
 6  post: cliente = cliente@pre
 7  post: fundos = fundos@pre
 8  post: ContaCorrente::allInstances() =
 9        ContaCorrente::allInstances()@pre
10 ...
```

Figura 7.13 – Exemplo de especificação da operação *depositar* contendo expressões associadas à definição do escopo das modificações permitidas.

Para tornar mais precisa a especificação em OCL das soluções admissíveis para uma operação, definimos um novo tipo de restrição que é especificada através da palavra reservada *modifiable.* Restrições desse tipo devem ser definidas no contexto de uma operação do modelo, e são avaliadas em conjunto com as pós-condições dessa operação. O objetivo desse tipo de restrição é definir explicitamente os elementos de um espaço de objetos que podem sofrer modificações em função da execução de uma operação. As modificações em um espaço de objetos podem ser classificadas em seis categorias:

- Criação de um ou mais objetos.
- Destruição de um ou mais objetos.
- Modificação no valor de um atributo, com escopo de instância, de um ou mais objetos.
- Modificação no valor de um atributo com escopo de classe.
- Criação de uma ligação entre objetos.
- Destruição de uma ligação entre objetos

Os elementos de uma restrição *modifiable* são especificados conforme a categoria de modificação. A seguir, são descritas as possíveis formas de definição desses elementos.

7.3.1 Nome de uma classe

Se um elemento de uma construção *modifiable* corresponder ao nome de uma classe *C* definida no modelo, a restrição define que a execução da operação pode resultar na criação ou na eliminação de objetos de *C* ou de classes descendentes de *C*. As pós-condições da operação podem definir restrições mais específicas, como por exemplo, as características esperadas para os objetos criados, a quantidade de objetos que deverão ser criados, quais objetos deverão ser destruídos.

Dessa forma, dado que existam, por exemplo, duas classes *A1* e *A2* definidas como descendentes de uma classe *A,* se a restrição *modifiable: A* for definida no contexto de uma operação *op*, isto significa que tanto a criação como a eliminação de instâncias de *A1*, *A2* e *A* (se *A* for uma classe concreta) são efeitos permitidos para essa operação.

Considerando que I_A seja o conjunto correspondente à união das instâncias da classe *A* e das instâncias das classes descendentes de *A* no espaço de objetos $\sigma(M)$ existente no instante de uma invocação de *op*, essa restrição define que a cardinalidade do conjunto I_A no espaço de objetos $\sigma'(M)$ obtido após a execução de *op* pode ser diferente da cardinalidade do conjunto I_A no espaço de objetos $\sigma(M)$.

A Figura 7.14 apresenta uma especificação para a operação *registrarNovoCliente* de um sistema bancário, na qual a restrição *modifiable: Cliente* (linha 2) define que as únicas modificações admissíveis para essa operação são a criação ou a eliminação de instâncias de Cliente. As pós-condições (linhas 3-7) especificam que o efeito esperado para essa operação corresponde à criação de uma única instância de Cliente, ou seja, nenhuma instância de Cliente pode ser eliminada pela operação.

```
1 context SistemaBanco::registrarNovoCliente()
2 modifiable: Cliente
3 def: novosClientes : Cliente =
4     Cliente::allInstances()->select(c | c.oclIsNew())
5 post: novosClientes->size() = 1
6 post: Cliente::allInstances() =
7       Cliente::allInstances()@pre->union(novosClientes)
8 ...
```

Figura 7.14 – Especificação da operação registrarNovoCliente

A Figura 7.15 apresenta uma especificação menos restritiva para essa mesma operação. A pós-condição (linha 3) define que uma e apenas uma nova instância de *Cliente* deve existir após a execução dessa operação, mas não há restrição em relação à eliminação de instâncias dessa classe que existam antes da execução dessa operação.

```
1 context SistemaBanco::registrarNovoCliente()
2 modifiable: Cliente
3 post: Cliente::allInstances()->one(c | c.oclIsNew())
```

Figura 7.15 – Especificação menos restritiva da operação registrarNovoCliente

7.3.2 Expressão OCL que resulte em um objeto ou em uma coleção de objetos

Uma expressão que resulte em um objeto ou em uma coleção de objetos, no contexto de uma restrição *modifiable*, estabelece que os valores de quaisquer atributos dos objetos definidos pela expressão podem ser modificados pela operação.

A Figura 7.16 apresenta a especificação da operação *depositar* da classe *ContaCorrente*, na qual a restrição *modifiable: self* (linha 2) estabelece que o valor de qualquer atributo do objeto que estiver recebendo a invocação da operação poderá ter um valor diferente após a execução dessa operação. A pós-condição (linha 3) restringe o valor esperado apenas para o atributo *saldo* desse objeto. Os demais atributos desse objeto poderão apresentar quaisquer valores após a execução dessa operação.

```
1 context ContaCorrente::depositar (valor : Real)
2 modifiable: self
3 post: saldo = saldo@pre + valor
```

Figura 7.16 – Especificação da operação depositar

A Figura 7.17 apresenta a especificação de uma operação que admite modificações nos valores de todos os atributos dos objetos da classe *ContaCorrente* recebidos através do parâmetro *contasAlvo*. A pós-condição (linha 4) estabelece o valor esperado para o atributo *saldo* das contas desse conjunto.

```
1 context SistemaBanco::aplicarTaxa(
2         contasAlvo : Set(ContaCorrente), taxa : Integer)
3 modifiable: contasAlvo
4 post: contasAlvo->forAll(c | c.saldo = c.saldo@pre – taxa)
```

Figura 7.17 – Especificação da operação aplicarTaxa

7.3.3 Expressão OCL do tipo *AttributeCallExp* referenciando um atributo com escopo de instância

Uma restrição *modifiable* definida por uma expressão que resulte em um objeto ou em uma coleção de objetos estabelece que os valores de quaisquer atributos desses objetos podem ser modificados em função da execução da operação. Para restringir a possibilidade de modificação a apenas um determinado atributo que tenha escopo de instância, deve ser utilizada uma expressão OCL do tipo *AttributeCallExp,* ou seja, uma expressão que referencie o atributo de um objeto ou de uma coleção de objetos determinada por uma expressão origem.

A Figura 7.18 apresenta uma nova versão para a especificação da operação *depositar* da classe *ContaCorrente*, na qual a restrição *modifiable: self.saldo* (linha 2) estabelece que apenas o valor do atributo *saldo* do objeto que estiver recebendo a invocação poderá ser alterado pela operação *depositar*. A pós-condição (linha 3) define o novo valor esperado para o atributo *saldo* desse objeto.

```
1 context ContaCorrente::depositar (valor : Real)
2 modifiable: self.saldo
3 post: saldo = saldo@pre + valor
```

Figura 7.18 – Especificação da operação depositar com a cláusula modifiable definida sobre um atributo específico

A Figura 7.19 apresenta uma nova especificação da operação *aplicarTaxa* que admite modificações apenas nos valores do atributo *saldo* dos objetos da classe *ContaCorrente* que tenham, no momento da invocação, *saldo* maior ou igual ao valor do parâmetro *valorMinimo*. A pós-condição (linha 4) estabelece o novo valor esperado para esse atributo em todas essas contas.

```
1 context SistemaBanco::aplicarTaxa(
2         contasAlvo : Set(ContaCorrente), taxa : Integer)
3 modifiable: contasAlvo.saldo
4 post: contasAlvo->forAll(c | c.saldo = c.saldo@pre – taxa)
```

Figura 7.19 – Especificação da operação aplicarTaxa com a cláusula modifiable definida sobre um atributo específico

Seja *σ(M),* o espaço de objetos correspondente ao instante de uma invocação da operação *op*, e *σ'(M)* o espaço de objetos obtido após a execução de *op.* Seja *ModifiedObjs* o conjunto de todos os objetos modificados pela execução de *op.* Seja

ModifiedAtts(obj) o conjunto de todos os atributos de um objeto *obj* que, após a execução de *op,* posuam um valor em *σ'(M)* diferente do que tinham em *σ(M).* Um objeto *obj* pertence ao conjunto *ModifiedObjs,* se *obj* existir tanto em *σ(M)* como em *σ'(M),* e *ModifiedAtts(obj)* não for o conjunto vazio.

Seja *ModifiableObjs* o conjunto definido por todos os objetos resultantes da avaliação das construções do tipo *modifiable* de *op* que resultem em um objeto ou em um conjunto de objetos. O conjunto *ModifiableObjs*, portanto, corresponde a todos os objetos que podem ter quaisquer de seus atributos modificados pela execução de *op.*

Seja *ModifiableAttExp* o conjunto de todas as expressões OCL do tipo *AttributeCallExp*, declaradas em construções *modifiable* de *op*, que referenciem um atributo com escopo de instância. Cada elemento desse conjunto tem a estrutura *source.att*, onde *source* corresponde a um conjunto de instâncias de um tipo *T,* e *att* corresponde a um atributo *att* de *T.*

Todo objeto *obj* do conjunto *ModifiedObjs* deve pertencer ao conjunto *ModifiableObjs* ou, então, para todo atributo *atrib* do conjunto *ModifiedAtts(obj),* deve existir um elemento *e* de *ModifiableAttExp* tal que *obj* pertença ao conjunto definido por *e.source* e *atrib* seja igual a *e.att*.

7.3.4 Expressão OCL do tipo *AttributeCallExp* referenciando um atributo com escopo de classe

Para definir a possibilidade de modificação do valor de um atributo que tenha escopo de classe, basta utilizar na construção *modifiable* uma expressão OCL do tipo *AttributeCallExp* que referencie esse atributo, conforme ilustrado pelo exemplo da Figura 7.20, onde a especificação da operação *alterarLimite* determina que apenas o valor do atributo *limite* da classe ContaCorrente pode ser modificado pela operação.

```
1 context SistemaBanco::alterarLimite(novoLimite : Real)
2 modifiable: ContaCorrente::limite
3 post: ContaCorrente::limite = novoLimite
```

Figura 7.20 – Especificação da operação alterarLimite

7.3.5 Criação e Destruição de Ligações entre Objetos

A definição das associações binárias que poderão ter ligações criadas ou destruídas em uma operação é realizada através de uma expressão da forma *links (origem, papel, destino)*, onde:

- *links* é uma palavra reservada adicionada à linguagem que define uma restrição de escopo associada à criação ou destruição de ligações entre objetos.
- *origem* corresponde a uma expressão OCL que resulta em um objeto ou em uma coleção de objetos de uma classe *A*.
- *papel* corresponde ao nome do papel de uma classe *B* em uma associação de *B* com *A*.
- *destino* corresponde a uma expressão que resulta em um objeto ou em uma coleção de objetos de *B*. A definição da expressão correspondente ao elemento *destino* é opcional. Se for omitida, assume-se que pode ser criada ou removida uma ligação de um objeto do conjunto *origem* com quaisquer instâncias de *B*.

Portanto, uma expressão *links* especifica que a operação admite, como um possível efeito gerado por sua execução, a criação ou destruição de ligações entre instâncias de uma classe *A* (determinadas pela expressão *origem)* e instâncias de uma classe *B* (determinadas pela expressão *destino*). Essas ligações são instâncias de uma associação entre *A* e *B*, onde o nome do papel de *B* nessa associação é igual a *papel.*

A definição das associações n-árias, onde n > 2, que poderão ter ligações criadas ou destruídas em uma operação é realizada através de uma expressão da forma *links (associação, lista_papel_destino)*, onde:

- *associação* corresponde ao nome da associação n-ária.
- *lista_papel_destino* corresponde a uma lista de elementos da forma *papel* : *destino*, separados por vírgulas, onde papel corresponde ao nome do papel de uma classe *C* dessa associação, e *destino* corresponde a uma expressão que resulta em um objeto ou em uma coleção de objetos de *C*. Se um papel de uma classe *C* definido na associação for omitido da lista, assume-se que não

há restrição em relação às instâncias de C nas ligações criadas ou removidas referentes a essa associação.

Nessa segunda forma, a expressão *links* especifica que uma operação pode criar ou remover ligações entre tuplas de objetos que pertençam ao produto cartesiano dos objetos especificados pelas expressões *destino* definidas em *lista_papel_destino.*

A Figura 7.21 apresenta alguns exemplos de definição de escopo relativa à criação ou destruição de ligações entre objetos. O caso A define que a operação *vincularFundos* pode criar ou destruir ligações entre quaisquer instâncias da classe *ContaCorrente* e quaisquer instâncias da classe *FundoInvestimento*. O caso B define que essa operação pode criar ou destruir ligações entre a instância *c* de ContaCorrente recebida como parâmetro e quaisquer instâncias da classe *FundoInvestimento.* No caso C, a operação pode criar ou destruir ligações entre a instância *c* de ContaCorrente recebida como parâmetro e quaisquer instâncias de *FundoInvestimento* que existirem no momento da chamada da operação. Finalmente, o caso D especifica que a operação pode criar ou destruir ligações entre a instância *c* de *ContaCorrente* recebida como parâmetro e as instâncias de *FundoInvestimento* também recebidas como parâmetro (*fundosAVincular)*.

```
Caso A:
1 context SistemaBanco::vincularFundos()
2 modifiable: links(ContaCorrente::allInstances(), fundos)

Caso B:
1 context SistemaBanco::vincularFundos(c : ContaCorrente)
2 modifiable: links(c, fundos)

Caso C:
1 context SistemaBanco::vincularFundos(c : ContaCorrente)
2 modifiable: links(c, fundos, c.fundos@pre)

Caso D:
1 context SistemaBanco::vincularFundos(c : ContaCorrente,
2              fundosAVincular : Set(FundoInvestimento))
3 modifiable: links(c, fundos, fundosAVincular)
```

Figura 7.21 - Exemplos de cláusulas modifiable aplicadas a ligações entre objetos

7.3.6 everything e nothing

Existem ainda duas palavras reservadas que podem ser utilizadas na especificação do escopo de modificações admissíveis para uma operação:

a) *everything*: qualquer elemento do espaço de objetos pode ser modificado. Esse é o valor definido como padrão para operações modificadoras, ou seja, se uma operação modificadora não definir nenhuma restrição do tipo *modifiable,* assume-se *everything.*

b) *nothing*: o espaço de objetos resultante da execução da operação deve ser igual ao espaço de objetos existente no instante da sua invocação. Esse é o valor definido como padrão para as operações de consulta, ou seja, se uma operação de consulta não definir nenhuma restrição do tipo *modifiable,* assume-se *nothing.*

7.4 Biblioteca padrão da OCL

Algumas operações definidas na biblioteca padrão da OCL possuem uma definição formal que não é compatível com a sua descrição informal. Em particular, a definição formal das operações *select* e *sum* pode gerar resultados incompatíveis com sua descrição informal nos casos em que a coleção contenha algum elemento com valor indefinido.

A definição informal da operação *select* é equivalente ao operador *select* da linguagem SQL, ou seja, o seu resultado corresponde ao subconjunto dos elementos para os quais a expressão de seleção resultar no valor verdadeiro, ou seja, os elementos para os quais a expressão resultar em falso, ou em um valor indefinido, não farão parte do subconjunto resultante.

Na especificação da OCL, a semântica da operação *select* para coleções do tipo *Set* é definida em função da operação *iterate*, conforme ilustrado na Figura 7.22. Para os demais tipos de coleção, a operação *select* é definida de forma análoga.

```
source->select(iterator | body) =
        source->iterate(iterator; result : Set(T) = Set{} |
               if   body
               then result->including(iterator)
               else result
               endif)
```

Figura 7.22 – Semântica da operação select conforme a especificação da OCL

A operação *iterate* percorre cada elemento da coleção, atualizando, em cada iteração, o seu resultado com o valor resultante da avaliação da expressão definida após a barra vertical. O valor inicial é definido pela expressão de inicialização associada à variável localizada antes da barra vertical. Portanto, o valor de uma expressão *iterate* é igual ao resultado obtido na sua última iteração.

O problema nessa definição da operação *select* ocorre quando a coleção *source* contém um ou mais elementos com valor indefinido. Como a iteração é baseada em uma expressão *if-then-else-endif*, se a expressão *body* resultar em um valor indefinido, o resultado da iteração será indefinido e, portanto, o resultado da operação *select* será indefinido.

Para tornar a definição da operação *select* compatível com a sua descrição informal, adotamos a definição descrita na Figura 7.23, segundo a qual, os elementos para os quais a expressão *body* não resulte no valor *verdadeiro* são ignorados.

```
source->select(iterator | body) =
        source->iterate(iterator; result : Set(T) = Set{} |
               if   not body.oclIsUndefined() and body
               then result->including(iterator)
               else result
               endif)
```

Figura 7.23 – Definição proposta para a operação select

A operação *sum* é definida para qualquer coleção como a soma dos valores de seus elementos, desde que a operação + esteja definida no tipo do elemento da coleção. Desta forma, a expressão `Set (MCCARTHY e HAYES, 1969)->sum()` resulta no valor 12. A interpretação aplicada para operações de agregação, na possível presença de elementos com valor indefinido, corresponde à soma dos valores dos elementos que tenham valor conhecido, ou seja, os elementos com valor indefinido são ignorados.

Entretanto, a definição da operação *sum* na especificação da OCL - `post: result = self->iterate(elem; acc : T = 0 | acc + elem)` - não equivale a essa interpretação. Como a soma de um valor indefinido com qualquer outro valor resulta num valor indefinido, o resultado da operação *sum* será sempre indefinido quando a coleção possuir algum elemento indefinido. A Figura 7.24 apresenta a definição semântica que empregamos para a operação *sum*, onde apenas os elementos de valor definido são computados na soma.

```
post: result = self->iterate(elem; acc : T = 0 |
                  if   elem.oclIsUndefined()
                  then acc
                  else acc + elem
                  endif)
```

Figura 7.24 – Definição proposta para a operação sum

Uma outra questão semântica ligada às operações de coleção da OCL está relacionada à definição das operações *exists* e *one.* A Figura 7.25 apresenta a definição da operação *exists* presente na especificação da OCL. A operação *source->exists(body)* resulta no valor verdadeiro, se a expressão *body* for verdadeira para pelo menos um elemento da coleção *source.* Caso a expressão *body* não seja verdadeira para nenhum elemento da coleção, o resultado será o valor falso, somente se *body* resultar no valor falso para todos os elementos da coleção. Caso contrário, o valor será indefinido, ou seja, não podemos afirmar se existe ou não algum elemento na coleção para o qual *body* seja verdadeiro.

```
source->exists(iterator | body) =
 source->iterate(iterator; result : Boolean = false | result or body)
```

Figura 7.25 – Definição da operação exists

A operação *one* é semelhante à operação *exists*, porém ela resulta no valor verdadeiro somente se a expressão *body* for verdadeira para um e apenas um elemento da coleção. Entretanto, como a operação *one* é definida em função da operação *select*, o seu resultado nunca será indefinido.

```
source->one(iterator | body) =
    source->select(iterator | body)->size() = 1
```

Figura 7.26 – Definição original da operação one

A Figura 7.26 apresenta a definição que adotamos para a operação *one*. De acordo com essa definição, se a coleção tiver algum elemento de valor indefinido, o resultado da operação será indefinido. Dessa forma, a operação *one* somente resulta no valor verdadeiro, se a expressão *body* for verdadeira para apenas um elemento da coleção e falsa para todos os demais elementos da coleção.

```
source->one(iterator | body) =
   let nTrue : Boolean =
     source->iterate(iterator; result : Integer = 0 |
        if  body
        then result + 1
        else result
        endif)
   in nTrue = 1
```

Figura 7.27 – Definição proposta para a operação one

Além das modificações na semântica de algumas operações definidas na biblioteca padrão da OCL, acrescentamos novos tipos primitivos e novas operações a alguns tipos definidos na biblioteca padrão da OCL, que são particularmente importantes para a elaboração de modelos de sistema de informação transacionais (CORREA e WERNER, 2004c).

Os tipos acrescentados à biblioteca são *Date*, para a representação e manipulação de datas, e *DateTime*, para a representação e manipulação de instantes de tempo. Várias operações foram adicionadas à classe String, uma vez que a especificação da OCL define apenas operações de substring e concatenação. A elas foram adicionadas operações de busca de substring, conversão para maiúsculas e minúsculas, comparação por expressões regulares, entre outras. Além disso, foram definidas novas operações de agregação para as coleções: *avg* (média dos elementos), *min* (elemento de menor valor) e *max* (elemento de maior valor). O apêndice A apresenta a definição das classes e operações adicionadas à biblioteca padrão.

7.5 Restrições e relações de generalização

Esta seção discute algumas limitações da especificação da OCL 2.0 relacionadas à semântica de certas construções no contexto de uma hierarquia de classes e apresenta propostas de extensão. Essas extensões são também particularmente relevantes nas reestruturações envolvendo movimentação de atributos para a classe ancestral.

A primeira limitação refere-se à definição do valor inicial de um atributo. De acordo com o metamodelo definido na especificação da OCL (Figura 7.28), uma expressão de inicialização deve estar associada a um atributo de uma classe, sendo que um atributo pode estar associado a somente uma expressão de inicialização. Dessa forma, dado um modelo *M,* no qual sejam definidas uma classe *A* com um atributo *atrib* com escopo de instância, e duas subclasses *A1* e *A2* de *A*. Nesse contexto, não é possível especificar que o valor inicial de *attrib* para as instâncias de *A1* é definido por uma expressão *x*, enquanto que, para as instâncias de *A2,* esse valor é definido por uma outra expressão *y*.

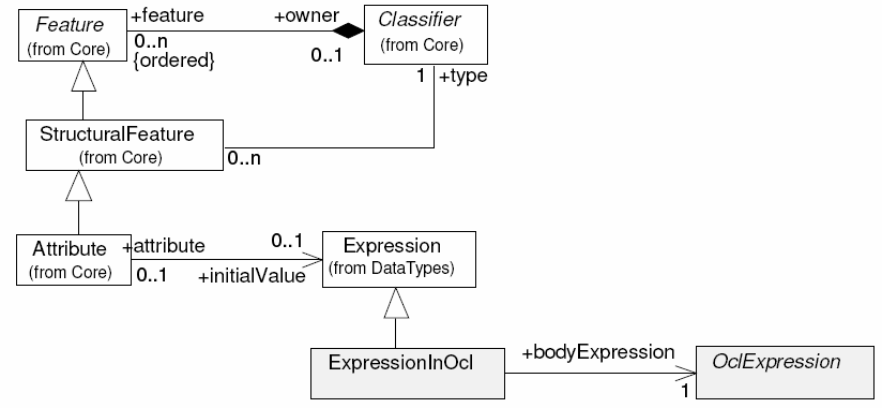
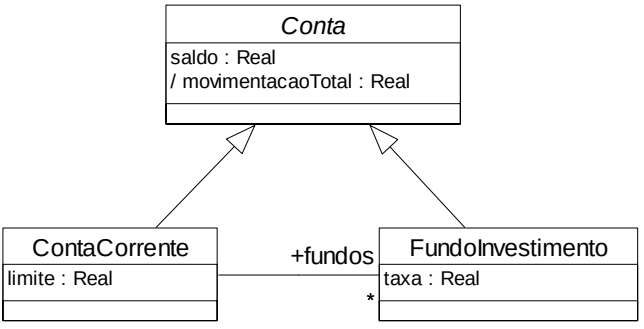


Figura 7.28 - Metamodelo OCL: definição de uma expressão de inicialização de atributo

A Figura 7.29 apresenta um diagrama de classes e duas expressões de inicialização para o atributo *saldo*. O valor inicial para esse atributo seria 100 para as instâncias da classe *ContaCorrente* e 500 para as instâncias da classe *FundoInvestimento*. Esse é um exemplo de especificação que não é permitida de acordo com o metamodelo apresentado na Figura 7.28.



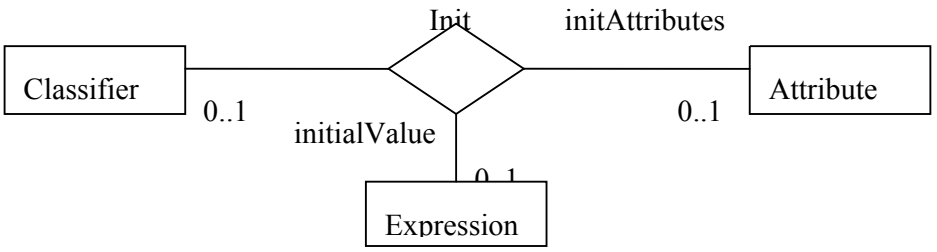

```
context ContaCorrente::saldo : Real
init: 100

context FundoInvestimento::saldo : Real
init: 500
```

Figura 7.29 –Definição de valores iniciais para um atributo em uma hierarquia de classes

A Figura 7.30 apresenta uma versão modificada do metamodelo, que permite a definição de diferentes valores iniciais para um atributo em uma hierarquia de classes. Nessa nova versão, uma expressão de inicialização é associada a uma tupla *Classifier-Attribute*, o que possibilita a definição das seguintes tuplas para o exemplo (ContaCorrente-saldo-100; FundoInvestimento-saldo-500).

A definição do valor inicial de um atributo segue a ordem de precedência definida pela hierarquia de classes, ou seja, prevalece a definição realizada na classe mais específica na hierarquia.

Um problema semelhante à definição do valor inicial de um atributo ocorre nas expressões de derivação do valor de um atributo. A definição formal da especificação OCL é omissa em relação à interpretação de uma expressão de derivação, enquanto que a descrição informal sugere uma conexão semelhante a do valor inicial, isto é, apenas uma expressão de derivação pode ser associada a um atributo de uma classe. Dessa forma, se existirem, em um modelo, duas classes *A1* e *A2* definidas como subclasses de uma classe *A,* e existir um atributo derivado *attrib* definido na classe *A*, não é possível definir expressões distintas de derivação para *attrib* em instâncias de *A1* e de *A2.*




```
-- só é permitido definir o valor inicial para atributos que sejam
-- definidos no classificador ou em algum ancestral do classificador
-- atributos derivados não podem estar associados a um valor inicial
context Classifier
inv: self.allAttributes->includesAll(initAttributes)
inv: self.initAttributes->forAll(a | not a.derived)

-- atributos com escopo de classe podem ter no máximo uma expressão de
-- inicialização.
context Attribute
inv: self.targetScope = ScopeKind::classifier implies
     initialValue->size() <= 1
```

Figura 7.30 - Metamodelo modificado para permitir diferentes inicializações em uma hierarquia de classes

Definimos uma expressão de derivação de forma análoga a uma expressão de valor inicial, isto é, uma expressão de derivação é associada a uma tupla <*Classificador*>-<*Atributo*>. Dessa forma, é possível definir expressões de derivação distintas para o atributo *movimentacaoTotal* definido na classe *Conta*, como mostra o exemplo da Figura 7.31. Esse atributo terá o valor definido pela expressão *saldo + fundos.saldo* para as instâncias da classe *ContaCorrente,* enquanto que para as instâncias de *FundoInvestimento*, o valor desse atributo será derivado da expressão *saldo*.

```
context ContaCorrente::movimentacaoTotal : Real
derive: saldo + fundos.saldo

context FundoInvestimento::movimentacaoTotal : Real
derive: saldo
```

Figura 7.31 – Expressões associadas a um atributo derivado em uma hierarquia de classes

7.6 OCL-AL (OCL Action Language)

Expressões OCL não podem produzir modificações no espaço de objetos onde elas são avaliadas e, portanto, elas podem ser utilizadas para especificar condições e restrições que devem ser respeitadas pela execução de uma ou mais ações, mas não para especificar as ações propriamente ditas (WARMER e KLEPPE, 2003).

A partir da versão 1.5, a especificação da UML passou a definir uma semântica de ações, a *UML Action Semantics* (UML AS), que permite a especificação de ações e atividades em um modelo de forma padronizada e independente de plataforma (OMG, 2003b). Embora a especificação da UML AS reconheça a necessidade de uma linguagem concreta, ela não define nem recomenda uma linguagem específica. A especificação apenas descreve um conjunto de mapeamentos dos elementos semânticos da UML AS nas linguagens concretas de ação como, por exemplo, a ASL (Action Specification Language) (KABIRA TECHNOLOGIES INC., 2006), a Kabira Action Semantics (KENNEDY CARTER LTD., 2002) e a BridgePoint Action Language (MELLOR e BALCER, 2002).

Existe, entretanto, uma significativa superposição entre as construções definidas na UML AS e aquelas definidas na especificação OCL. Ações de leitura, ações primitivas, invocação de operações de consulta, acesso a propriedades de objetos, acesso a ligações entre objetos, bem como operações de acesso e manipulação de coleções são alguns exemplos de ações que podem ser expressas em OCL, uma vez que nenhuma dessas ações é capaz de produzir modificações no espaço de objetos onde elas são executadas. De forma geral, as linguagens concretas de ação para modelos UML definem sintaxes específicas para essas ações, não utilizando as construções definidas na OCL.

Explorando a superposição existente entre a OCL e a UML AS, definimos a linguagem denominada OCL-AL (OCL Action Language). A OCL-AL é uma linguagem que estende a OCL de forma a permitir a especificação de ações sobre instâncias de modelos no nível M0 (instâncias de um modelo UML, por exemplo), bem como de instâncias de modelos no nível M1 (instâncias de um meta-modelo UML, por exemplo).

O principal objetivo da OCL-AL é possibilitar a especificação explícita de ações associadas a operações tanto de modelos M1 como de modelos M2, de forma a permitir

a elaboração de modelos executáveis, bem como a execução de operações de manipulação e transformação aplicáveis a esses modelos como, por exemplo, reestruturações e aplicação de padrões.

7.6.1 Sintaxe Abstrata

Ação é a unidade fundamental de especificação de comportamento. Uma ação recebe um conjunto de entradas e gera um conjunto de saídas, não sendo obrigatória a presença de elementos nesses conjuntos. Além disso, algumas ações modificam o espaço de objetos sobre o qual elas são executadas. Em síntese, as ações podem ser divididas nas seguintes categorias:

- Ações de Leitura: ações que são utilizadas para acessar os valores de variáveis, de atributos de objetos e de ligações entre objetos, sem alterá-los.
- Ações de Escrita: ações que produzem modificações no espaço de objetos ou em variáveis. Exemplos de ações desta categoria: criar um objeto, destruir um objeto, modificar o valor de uma variável; modificar o valor de uma propriedade de um objeto; criar uma ligação entre objetos, destruir uma ligação entre objetos.
- Funções Primitivas: transformam um conjunto de valores de entrada em um conjunto de valores de saída. Essas funções dependem exclusivamente dos valores de entrada, não produzindo nenhum efeito além da geração dos valores de saída, isto é, elas não acessam nem tampouco alteram objetos e ligações entre objetos. Funções aritméticas e funções booleanas são exemplos de funções primitivas.
- Ações de Invocação: disparam ações síncronas ou assíncronas, tais como chamadas de operações e envio de sinais para um ou mais objetos alvo.

Ações estão contidas em atividades que fornecem o seu contexto. Uma atividade especifica a coordenação de execução de uma ou mais ações subordinadas através de estruturas de controle e fluxos de dados. Através das atividades, é possível definir estruturas de seqüência, iteração e condição.

A OCL-AL é uma extensão da OCL que possibilita a especificação de ações que possam modificar o espaço de objetos sobre as quais elas sejam executadas, além de

estruturas de controle e fluxo. As ações de leitura, funções primitivas e invocações de operações de consulta são expressas em OCL. A OCL-AL foi definida a partir da integração dos elementos que definem a semântica da OCL com os elementos definidos na UML AS.

A Figura 7.32 apresenta os principais elementos que definem as ações na UML AS e que foram reutilizados na definição da OCL-AL.

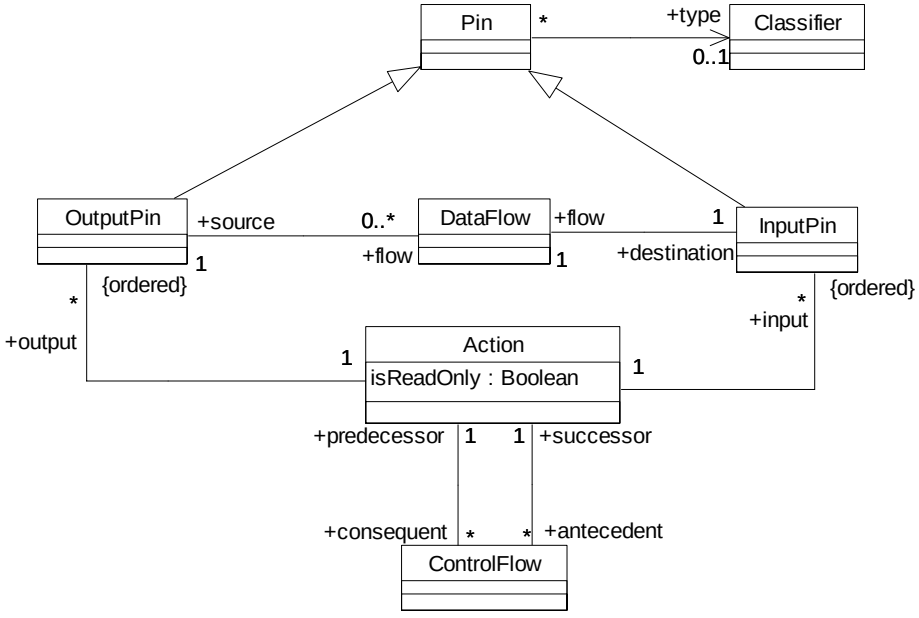


Figura 7.32 – OCL-AL: Principais elementos relacionados à definição de uma ação

Uma ação (*Action)* possui um conjunto ordenado de pinos de entrada e saída (*InputPin* e *OutputPin*). Cada pino pode conter valores que devem estar em conformidade com o classificador (*Classifier)* associado. Um fluxo de dados (*DataFlow)* conecta um pino de entrada de uma ação (*destination)* a um pino de saída de outra ação (*source)*, estabelecendo uma ordem implícita na execução das ações. Um fluxo de controle (*ControlFlow)*, por sua vez, impõe uma ordem explícita de execução entre um par de ações.

As ações básicas de criação de um objeto, destruição de um objeto e de escrita de valores em atributos de objetos são apresentadas no diagrama da Figura 7.33. *CreateObjectAction* resulta na criação de uma instância do classificador associado no espaço de objetos, que é colocada no pino de saída (*result*). *DestroyObjectAction* corresponde à ação de eliminação do objeto presente no pino de entrada (*input*).

WriteAttributeAction é a superclasse correspondente às ações que modificam o valor de um atributo de um objeto presente no pino de entrada (*object)*. Se esse pino não contiver nenhum objeto, o atributo associado deve ter escopo de classe. O valor a ser escrito é dado no pino de entrada *value*. A adição de valores a um atributo é realizada através da ação *AddAttributeValueAction*, que tem a opção de remover todos os valores anteriormente presentes, definida pelo atributo *isReplaceAll*.

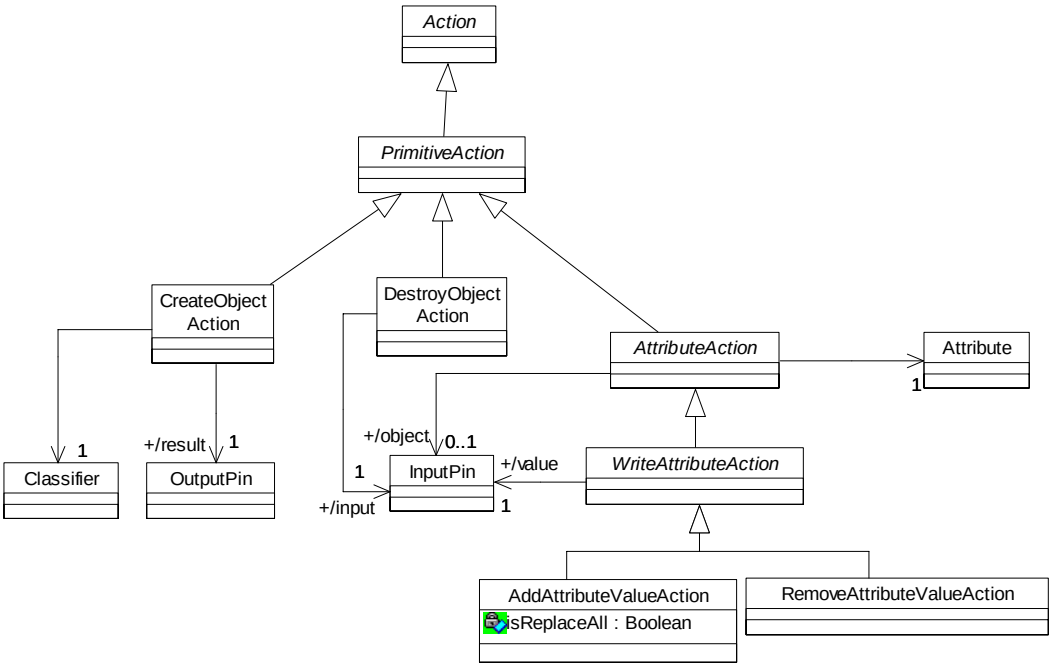


Figura 7.33 – OCL-AL: Ações de Objeto e de Atributo

As ações de criação e destruição de ligações entre objetos são representadas pelo diagrama apresentado na Figura 7.34. *CreateLinkAction* corresponde à ação de criação de um ligação entre objetos. Os objetos a serem ligados são definidos pelos pinos de entrada (*value*) associados às instâncias de *LinkEndCreationData* correspondentes à ação de criação de ligação (*CreateLinkAction*). A ação *CreateLinkObjectAction* corresponde à criação de uma ligação entre objetos cuja associação é uma classe associativa. O objeto associativo criado é retornado no pino de saída *result.* A classe *DestroyLinkAction* corresponde à ação de eliminação de uma ligação entre objetos. Os objetos participantes da ligação são definidos da mesma forma como em *CreateLinkAction*. A destruição de uma ligação entre objetos correspondente a uma classe associativa implica na destruição do respectivo objeto associativo.

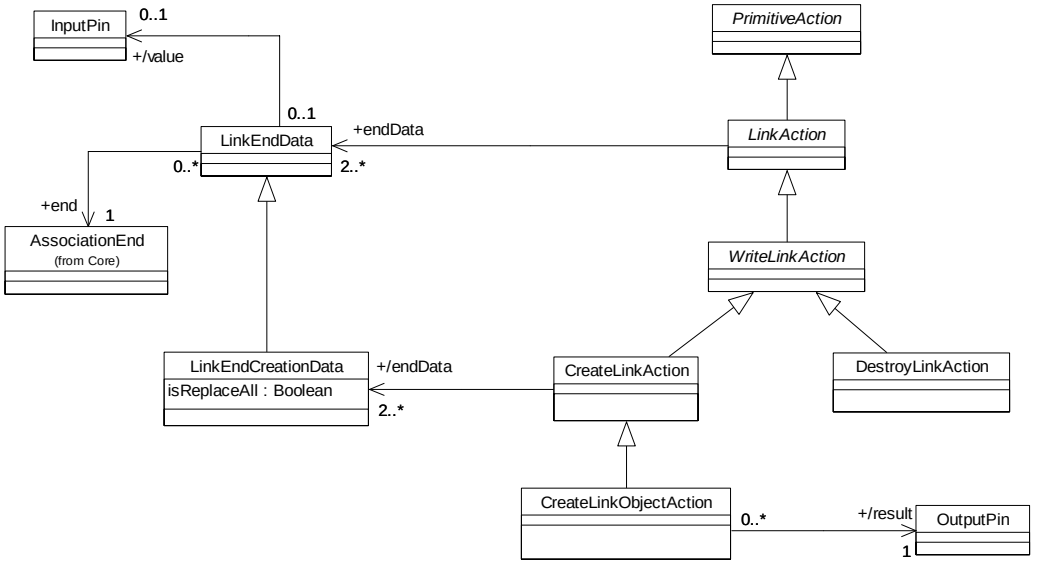


Figura 7.34 – OCL-AL: Ações de Criação e Destruição de Ligações entre Objetos

A Figura 7.35 apresenta o diagrama de classes correspondente à ação de chamada de uma operação que pode modificar o estado do sistema (*CallOperationAction)*, isto é, operações definidas com o atributo *isQuery* contendo o valor *false.* O objeto que recebe a chamada é definido pelo pino de entrada *target.* Os argumentos da operação são definidos por pinos de entrada (*argument)*, e os resultados da execução da operação são liberados nos pinos de saída (*result)* associados à ação (*ExplicitInvocationAction)*.

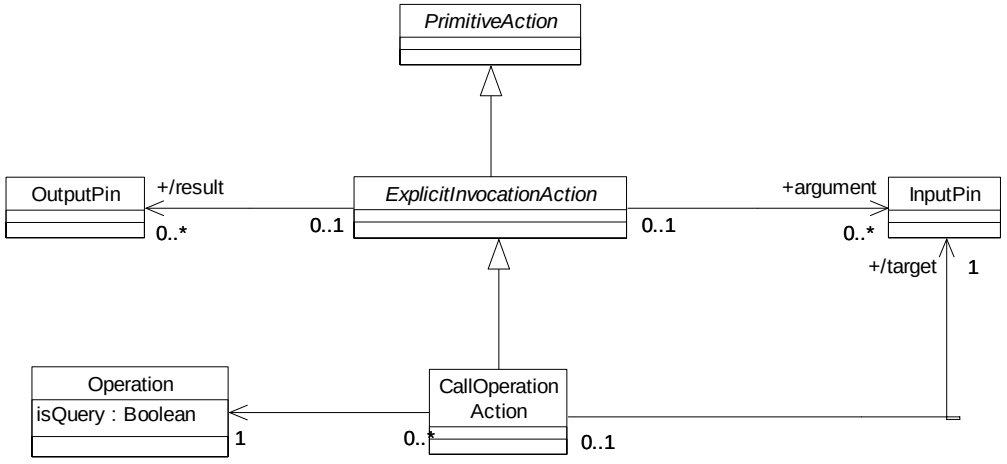


Figura 7.35 – OCL-AL: Ação de chamada a uma operação modificadora

Um grupo de ações (*GroupAction*) define uma composição de ações (*subAction*), como pode ser observado no diagrama de classes da Figura 7.36.

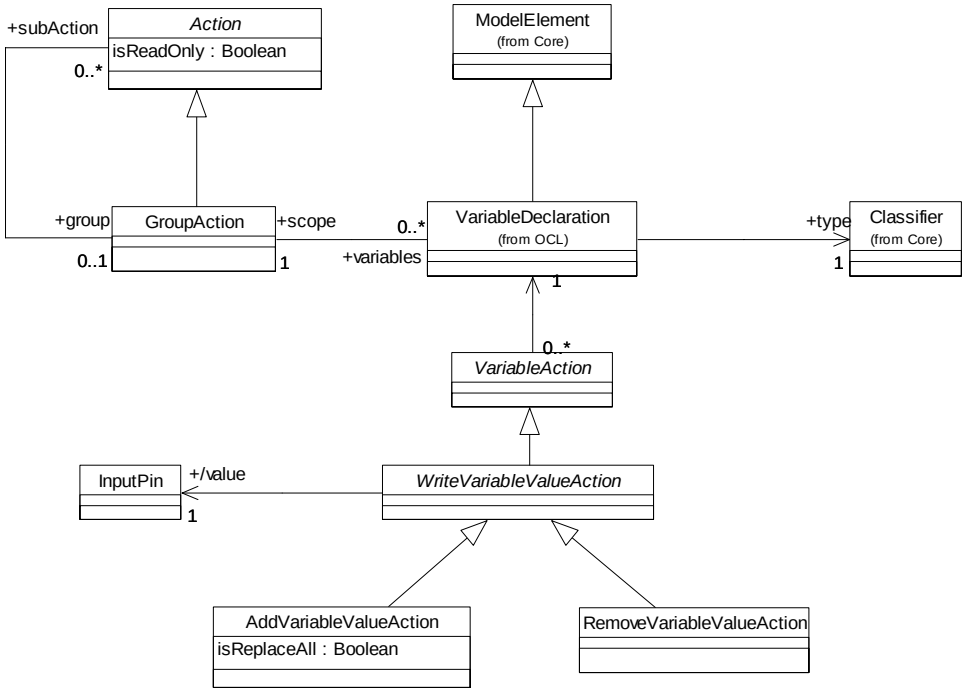


Figura 7.36 – OCL Action Language: Bloco de Ações

A seqüência das ações é definida pelos fluxos de dados e de controle que as conectam. Os pinos de entrada de um grupo de ações correspondem a todos os pinos de entrada das suas ações subordinadas que não estejam conectadas a outras ações do mesmo grupo. Os pinos de saída de um grupo correspondem a todos os pinos de saída de todas as ações subordinadas, uma vez que um pino de saída pode ser origem para vários fluxos de dados. Grupos de ações podem ser utilizados em fluxos de controle, de forma a possibilitar a sincronização da execução de grupos de ações.

Variáveis locais (*VariableDeclaration)* podem ser definidas em um grupo de ações e seus valores podem ser utilizados em fluxos de dados entre ações. Uma variável local é um elemento capaz de armazenar valores que são acessíveis apenas pelas ações definidas dentro de um grupo. A saída de uma ação pode ser escrita em uma variável e utilizada como entrada em uma ação subseqüente, possibilitando um caminho indireto de comunicação entre ações. As ações de modificação de uma variável são dadas pelas subclasses de *WriteVariableValueAction*. A classe *AddVariableValueAction* corresponde à adição de um valor, dado pelo pino de entrada *value,* à variável associada

à ação. O atributo *isReplaceAll* indica se os valores previamente existentes na variável devem ser removidos. *VariableDeclaration* é uma classe definida no meta-modelo da OCL e que foi reutilizada no contexto de definição de variáveis locais de um grupo de ações, uma vez que elas podem estar associadas a ações de leitura dos valores dessas variáveis, e essas ações são definidas através de expressões OCL.

Uma ação condicional (Figura 7.37) possibilita a execução de ações condicionada ao resultado de ações de teste. Uma ação condicional consiste em um conjunto de cláusulas (instâncias da classe *Clause*), sendo que cada cláusula possui uma ação de teste (*test)* e um corpo (*body)*. É possível definir uma ordem de precedência entre as cláusulas que compõem uma ação condicional. Se essa ordem não for definida, as ações de teste podem ser executadas de forma concorrente. Cada cláusula determina um pino de saída para sua ação de teste. Se o valor desse pino for *true*, o corpo é executado. O valor do pino de saída de uma ação condicional será o valor do pino de saída correspondente ao corpo executado.

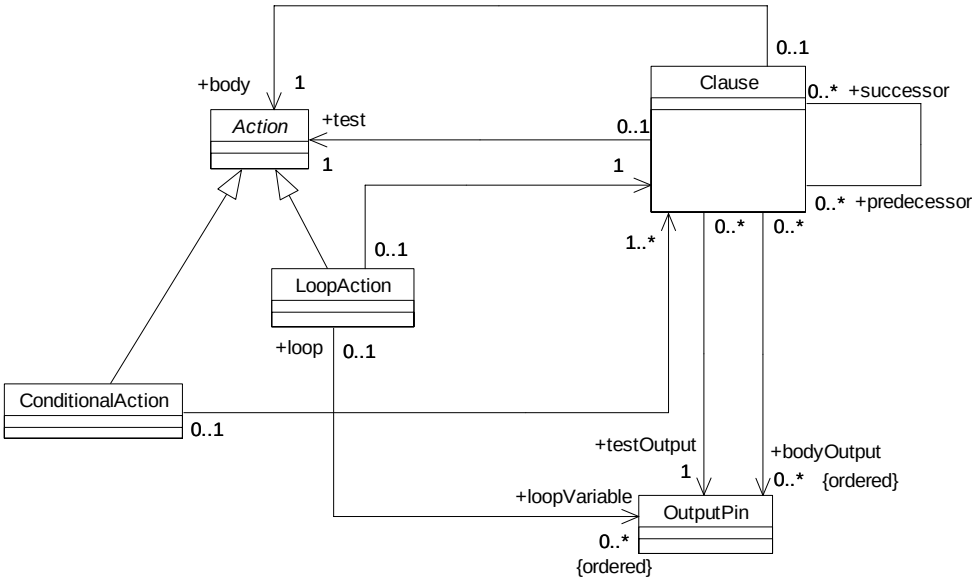


Figura 7.37 – Ocl Action Language: Laço e Ações Condicionais

Uma ação de repetição (laço) contém uma cláusula com uma ação de teste (*test)* e uma ação denominada corpo do laço (*body)*. O corpo do laço é executado repetidamente enquanto a ação de teste resultar no valor *true.* A ação de teste e o corpo do laço têm acesso aos valores de um conjunto de variáveis do laço, que são representadas como um conjunto ordenado de pinos de saída associados à ação de

repetição. As variáveis do laço podem ser conectadas aos pinos de entrada da ação de teste e do corpo do laço, fornecendo os valores correntes dessas variáveis durante a execução do laço.

Uma ação (ou grupo de ações) pode ser aplicada a todos os elementos de uma coleção. A Figura 7.38 apresenta os elementos envolvidos nessa construção. *CollectionAction* corresponde à aplicação de uma ação (*subAction)* a todos os elementos de uma coleção recebida em um pino de entrada (*collection)*. Uma ação de coleção corresponde a um laço onde, em cada iteração, o pino de entrada *loopVariable* é carregado com o próximo elemento da coleção e a ação *subAction* é executada.

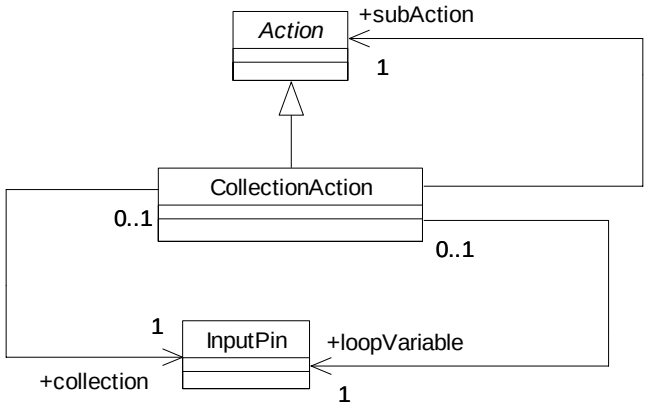


Figura 7.38 – Ocl Action Language: Ação aplicada a uma coleção

A integração das ações que provocam mudanças no espaço de objetos onde elas são executadas com expressões OCL é ilustrada no diagrama da Figura 7.39. Ao invés das expressões OCL serem mapeadas em ações de leitura, funções primitivas e chamadas de operações, definimos uma ação primitiva correspondente à avaliação de uma expressão OCL. Essa ação é denominada *OclExpressionEvalAction* e está associada a uma expressão OCL (*OclExpression*). O resultado da avaliação dessa expressão OCL é gerado no pino de saída da ação, denominado *result*. A opção por essa forma de integração teve como objetivo reutilizar toda a semântica de avaliação de expressões definida na especificação da OCL, bem como a infra-estrutura disponível para avaliação de expressões OCL no contexto de uma máquina virtual de execução de ações em modelos.

As ações que não alteram o espaço de objetos foram mapeadas da seguinte forma: Uma ação de leitura de valores de atributos de objetos corresponde à avaliação de uma expressão OCL do tipo *AttributeCallExp*. Uma ação de leitura de ligação entre objetos é realizada pela avaliação de uma expressão OCL do tipo *AssociationEndCallExp*. A classe *LiteralExp* corresponde a expressões literais que podem ser utilizadas para carregar pinos de entrada em funções primitivas, em chamadas de operações, ou em outras ações. Funções primitivas são mapeadas em chamadas a operações (*OperationCallExp*) de elementos primitivos como, por exemplo, Boolean, Integer, Real e String, definidos na especificação da OCL.

Dois elementos receberam um tratamento especial por estarem presentes tanto na semântica de ações como na definição da OCL: acesso a variáveis e chamadas de operações. As variáveis definidas em expressões OCL através da construção *let* não podem ser referenciadas em ações de escrita, ou seja, seu valor não pode ser alterado. Entretanto, as variáveis definidas em um grupo de ações podem ser acessadas por expressões OCL (*VariableExp*).

Chamadas de operações de consulta, isto é, operações definidas com o atributo *isQuery = true*, são mapeadas para instâncias da classe *OperationCallExp* definida na OCL, enquanto que aquelas definidas com o atributo *isQuery = false* são mapeadas como instâncias da classe *CallOperationAction*, conforme descrito anteriormente.

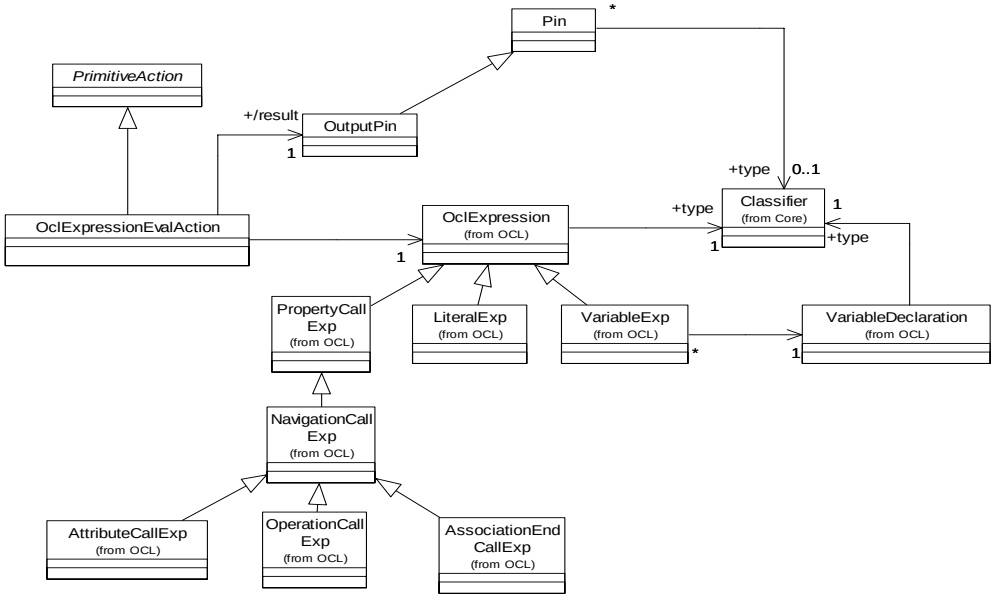


Figura 7.39 – OCL-AL: Integração entre o meta-modelo da OCL e a semântica de ações.

7.6.2 Sintaxe Concreta

A sintaxe concreta da OCL-AL foi definida como uma extensão da sintaxe concreta da OCL. As ações são especificadas no contexto de operações modificadoras do modelo subjacente. Enquanto a especificação explícita de uma operação de consulta é realizada através da palavra reservada *body* associada a uma expressão OCL, a especificação explícita de uma operação modificadora é definida pela palavra reservada *actionBody* associada a uma ação ou a um grupo de ações. Um grupo de ações é delimitado pela construção *begin – end*, sendo o ponto e vírgula utilizado como separador das ações de um grupo.

Os tipos utilizados na OCL-AL são os mesmos tipos definidos na especificação da OCL somados aos tipos definidos no modelo subjacente. Variáveis e constantes podem ser definidas em um grupo de ações, conforme ilustrado pelo exemplo da Figura 7.40. Nesse exemplo, são definidas duas variáveis do tipo *Integer*, *qtde* e *noEmprestimos*, e uma constante *tituloPadrao* do tipo *String*.

```
var   qtde : Integer;
var   noEmprestimos: Integer = 0;
const tituloPadrao : String = 'Mr.';
```

Figura 7.40 - OCL-AL: Exemplos de definições de variáveis e constantes

Objetos são criados pelo comando *create*. Os valores iniciais para os atributos do objeto criado podem ser definidos, opcionalmente, em um comando *create.* Objetos são destruídos através do comando *delete,* sendo que todas as ligações do objeto destruído com outros objetos também são destruídas. Ações de escrita que atualizam variáveis, atributos e ligações entre objetos são geradas pelo comando de atribuição, que tem a forma *lhs := rhs.*

A Figura 7.41 ilustra alguns exemplos de utilização desses comandos. A criação de ligações entre objetos é feita através de um comando de atribuição onde a parte *lhs* corresponde a uma expressão do tipo *AssociationEndCallExp,* definido na OCL. Entre a classe *Aluguel* e *Cliente* existe uma associação na qual a multiplicidade de *Cliente* é 1 e de *Aluguel* é *. O comando de atribuição especificado na linha 13 é traduzido numa ação que cria uma ligação entre os objetos *umAluguel* e *umCliente,* e remove qualquer ligação porventura existente entre *umAluguel* e outro objeto *Cliente*.

Ligações entre objetos também podem ser modificadas através de um comando de atribuição envolvendo o papel correspondente a um lado da associação com multiplicidade *muitos (*)*. No exemplo da Figura 7.41, existe uma associação entre as classes *Aluguel* e *ItemAluguel*, onde a multiplicidade de *Aluguel* é 1 e *ItemAluguel* é *. A parte *lhs* da atribuição presente na linha 22 desse exemplo corresponde aos *itens* ligados ao objeto *umAluguel*. A parte *rhs* dessa atribuição corresponde ao conjunto dos *itens* ligados ao objeto *umAluguel,* incluindo-se o objeto *umItemAluguel*. O efeito de uma atribuição *obj.role* := *rhs,* onde *role* é o papel correspondente a um lado com multiplicidade maior que 1 de uma associação, é o seguinte: para todo elemento da coleção *obj.role* que não faça parte da coleção *rhs,* é eliminada a ligação entre esse elemento e *obj*, e, para todo elemento da coleção *rhs* que não faça parte da coleção *obj.role*, é criada uma ligação entre esse elemento e *obj.* Na ação especificada na linha 22 do exemplo, portanto, o efeito será a criação de uma ligação entre os objetos *umAluguel* e *umItemAluguel*.

```
 1  var umAluguel : Aluguel;
 2  var umItemAluguel : ItemAluguel;
 3  var umCliente : Cliente;
 4
 5 // criação de um objeto Aluguel
 6 umAluguel := create Aluguel;
 7
 8 // ação de escrita – atributo numero do objeto Aluguel
 9 umAluguel.numero := 10;
10
11 // ação de criação de uma ligação entre os objetos umAluguel e
12 // umCliente
13 umAluguel.cliente := umCliente;
14
15 // criação de um objeto ItemAluguel, definindo os valores para
16 // seus atributos taxa e retornoEsperadoPara
17 umItemAluguel := create ItemAluguel( taxa := 10.0,
18                       retornoEsperadoPara := dataRetorno);
19
20 // criação de uma ligação entre os objetos umAluguel e
21 // umItemAluguel
22 umAluguel.itens := umAluguel.itens->including(umItemAluguel);
```

Figura 7.41 – OCL-AL: Exemplos de ações de escrita

Repetições de ações podem ser definidas através das seguintes construções: *while* <*condition*> *do* <*statement*>, *repeat* <*statement*> *until* <*condition*> e *for* <*var*>*:=* <*expression1*> *to* <*expression2*> *step* <*expression3*> *do* <*statement*>. No comando *for,* expression_i corresponde a expressões OCL do tipo Integer. Os comandos *break* e *continue* podem ser utilizados, respectivamente, para sair do bloco de repetição e pular para a condição de repetição.

A aplicação de uma ação ou grupo de ações a todos os elementos de uma coleção pode ser realizada através da construção *foreach* <*iterator*> [*:* <*type*>] *in* <*collection*> *do* <*statement*>. As ações definidas em <*statement*> são executadas para cada elemento da coleção <*collection*>, que é definida por uma expressão OCL. O elemento da coleção é referenciado pelas ações através da variável <*iterator*>.

A OCL define a construção condicional *if-then-else-endif* que corresponde a uma expressão cujo valor é dado pela expressão *then* ou pela expressão *else*, dependendo da avaliação da expressão de condição (*if*). Preservando a semântica da construção condicional já presente na OCL, a OCL-AL define a construção *doif-then-else* para a execução condicional de ações.

A invocação de uma operação é expressa da mesma forma tanto para as operações que possam modificar o espaço de objetos, como para as operações de consulta. Conforme descrito anteriormente, chamadas a operações de consulta são mapeadas para expressões OCL de chamada de operação (*OperationCallExp*), enquanto que as chamadas para operações modificadoras são mapeadas para ações de invocação de operação (*CallOperationAction*). O retorno de uma operação é definido pela construção *return [exp],* onde *exp* corresponde a uma expressão cujo valor será retornado pela operação.

7.7 Considerações Finais

Este capítulo apresentou algumas extensões à OCL para lidar com questões como: a impossibilidade de definir diferentes expressões de inicialização e derivação para um atributo em uma hierarquia de classes; inconsistências entre as definições informal e formal de algumas operações da biblioteca padrão da OCL; ausência de uma forma de especificar o escopo das modificações admissíveis por uma operação do modelo. A solução proposta para esta última questão consiste na utilização de uma

construção (*modifiable*), que permite definir as modificações que uma operação pode gerar em um espaço de objetos. Essa solução é similar às construções existentes em outras linguagens de especificação, como VDM, JML, entre outras.

Existem outras questões ligadas à semântica da OCL que não foram abordadas, como a interpretação de expressões OCL no contexto de diagramas de estados, uma vez que nos limitamos a utilizar os elementos da UML e da OCL para os quais existe uma definição formal, descrita no apêndice da especificação da OCL (OMG, 2003a).

Ao contrário das linguagens criadas para a elaboração de modelos UML executáveis, baseadas na UML Action Semantics, que definem construções específicas para as ações de leitura, as quais poderiam ser expressas em OCL, definimos uma linguagem de ação (OCL-AL) que reutiliza a sintaxe e a infra-estrutura disponível para a OCL. Dessa forma, as operações, tanto no nível M1 como no nível M2, podem ser especificadas através de uma combinação da forma explícita (OCL-AL) com a forma implícita (OCL), de modo similar ao oferecido em linguagens como VDM++, por exemplo.

O próximo capítulo descreve o software Odyssey-PSW, implementado no contexto desta tese, que oferece apoio automatizado para a realização de reestruturações através da execução automática de testes de regressão, seguindo a abordagem descrita no capítulo 6. Além disso, todas as extensões propostas neste capítulo também foram implementadas nesse software.

Capítulo 8

Odyssey-PSW

8.1 Introdução

Este capítulo descreve o software denominado Odyssey-PSW, cujo propósito é apoiar algumas atividades de verificação e validação de um modelo com restrições especificadas em OCL, bem como a avaliação da preservação da semântica do modelo após reestruturações, conforme abordagem descrita no capítulo anterior. O software Odyssey-PSW foi projetado de forma a complementar os recursos de modelagem de software oferecidos pelos diversos ambientes e ferramentas de modelagem disponíveis atualmente. Odyssey (reuse.cos.ufrj.br/odyssey), IBM Rational Rose (www.ibm.com/software/rational), Borland Together (www.borland.com/together), Poseidon (www.gentleware.com) e Jude (jude.esm.jp) são alguns exemplos dessas ferramentas. Os elementos do modelo, e.g., classes, atributos, operações e associações, devem ser definidos em uma ferramenta externa de modelagem, enquanto que a elaboração e a avaliação das especificações OCL/OCL-AL que complementam o modelo são realizadas através do Odyssey-PSW.

O restante deste capítulo descreve as principais funcionalidades do Odyssey-PSW, e está organizado nas seguintes seções: a seção 8.2 descreve a organização de um modelo no Odyssey-PSW e a sua integração com as ferramentas externas de modelagem; a seção 8.3 descreve as principais características do editor e compilador de especificações OCL/OCL-AL; a seção 8.4 descreve o gerenciador de espaços de objetos, módulo que possibilita a criação de instâncias e ligações entre as instâncias das classes do modelo e a avaliação das restrições gerais do modelo sobre essas instâncias; a seção 8.5 descreve como o software apóia a definição de casos de teste para especificações explícitas de operações e de expressões OCL associadas à definição de atributos derivados; a seção 8.6 descreve o apoio à definição de casos de teste para especificações implícitas de operações; a seção 8.7 apresenta um exemplo de como as construções OCL do modelo podem indicar pontos para reestruturação do modelo, e

como os casos de teste de operações de consulta e modificadoras podem apoiar essa reestruturação. Finalmente, a seçao 8.8 apresenta as considerações finais deste capítulo.

8.2 Definição de um Projeto

No Odyssey-PSW, um modelo é organizado em um projeto composto por duas partes:

- Um arquivo XMI contendo as definições do modelo, por exemplo, classes, atributos, operações, associações, etc. A definição dos elementos de um modelo deve ser realizada utilizando-se uma ferramenta de modelagem de software capaz de exportar o modelo produzido para o formato XMI (XML Metadata Interchange) (OMG, 2002b). XMI é um padrão recomendado pelo OMG (Object Management Group) que estabelece um formato baseado na XML (eXtensible Markup Language) para a descrição dos elementos de um modelo, com o objetivo permitir o intercâmbio de metadados entre as diversas ferramentas de modelagem.
- Um conjunto de arquivos fonte contendo as definições OCL/OCL-AL associadas aos elementos do modelo. Optou-se por realizar essas definições em arquivos texto à parte pelos seguintes motivos: a maioria das ferramentas de modelagem não oferece recursos para a edição de restrições OCL; dificuldade para intercambiar as definições OCL elaboradas nas ferramentas que oferecem tais recursos, visto que a maioria não exporta as restrições no formato XMI; oferecer maior agilidade aos usuários durante o processo de edição e verificação das expressões OCL, uma vez que não é necessário importar as definições do modelo a cada modificação efetuada em uma determinada expressão.

A Figura 8.1 apresenta um extrato da primeira versão do modelo referente a um sistema para logística de transporte ferroviário que será utilizado nos exemplos apresentados neste capítulo. Este modelo foi elaborado na ferramenta *Rational Rose* 2003, e suas classes estão organizadas em dois pacotes: *Recursos* e *CCP*. No pacote *Recursos*, estão presentes as classes relacionadas aos recursos ferroviários que podem estar estacionados em uma linha: *Locomotiva*, *Vagão* e *Trem*. Um trem deve ser formado, no mínimo, por uma locomotiva e um vagão. Todo trem tem origem em um pátio e destino em um outro pátio. Cada pátio possui uma ou mais linhas onde os

recursos podem estar estacionados. As classes *Pátio* e *Linha* estão definidas no pacote *CCP (Centro de Controle de Pátios)*. Este modelo pode ser exportado para um arquivo no formato XMI através do software *Unisys XML plug-in,* acoplável ao *Rational Rose*.

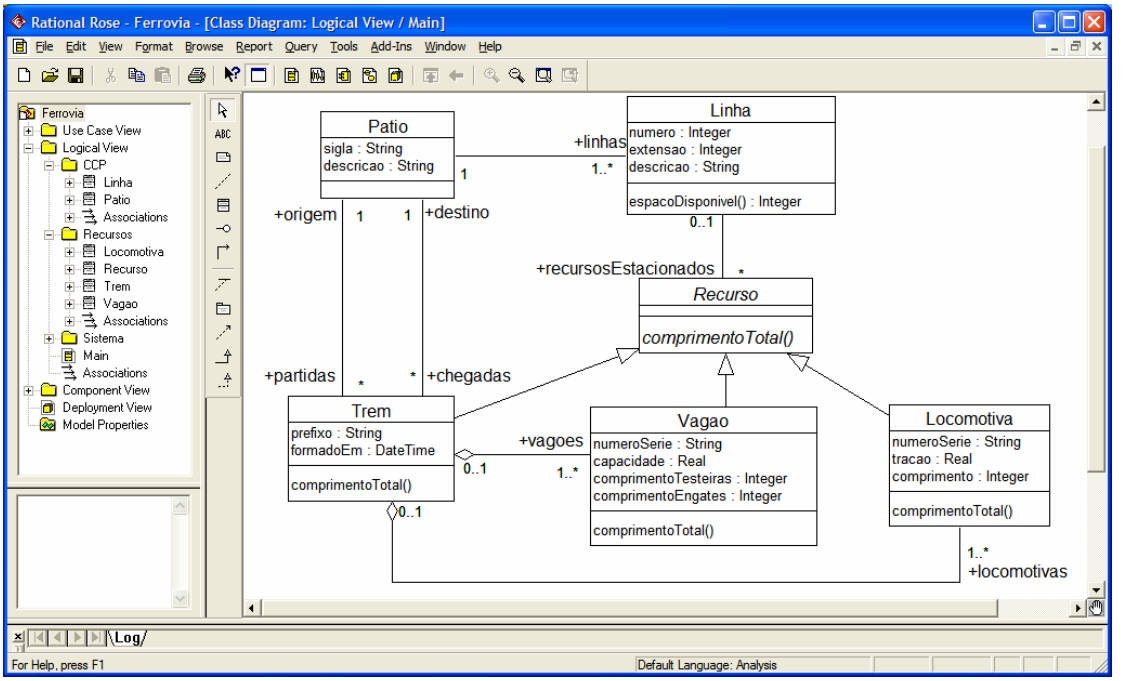


Figura 8.1 - Extrato de um modelo produzido em uma ferramenta CASE externa

Uma vez que o arquivo contendo os elementos do modelo no formato XMI esteja disponível, um projeto pode ser criado no Odyssey-PSW, informando-se o nome deste arquivo e a relação de arquivos OCL que compõem o projeto, caso estes já estejam disponíveis. Após a definição da relação inicial dos arquivos que compõem o novo projeto, os elementos do modelo presentes no arquivo XMI são importados para o Odyssey-PSW e apresentados em uma estrutura hierárquica. A Figura 8.2 apresenta a janela resultante da importação do modelo da Figura 8.1 presente no arquivo Ferrovia.xml.

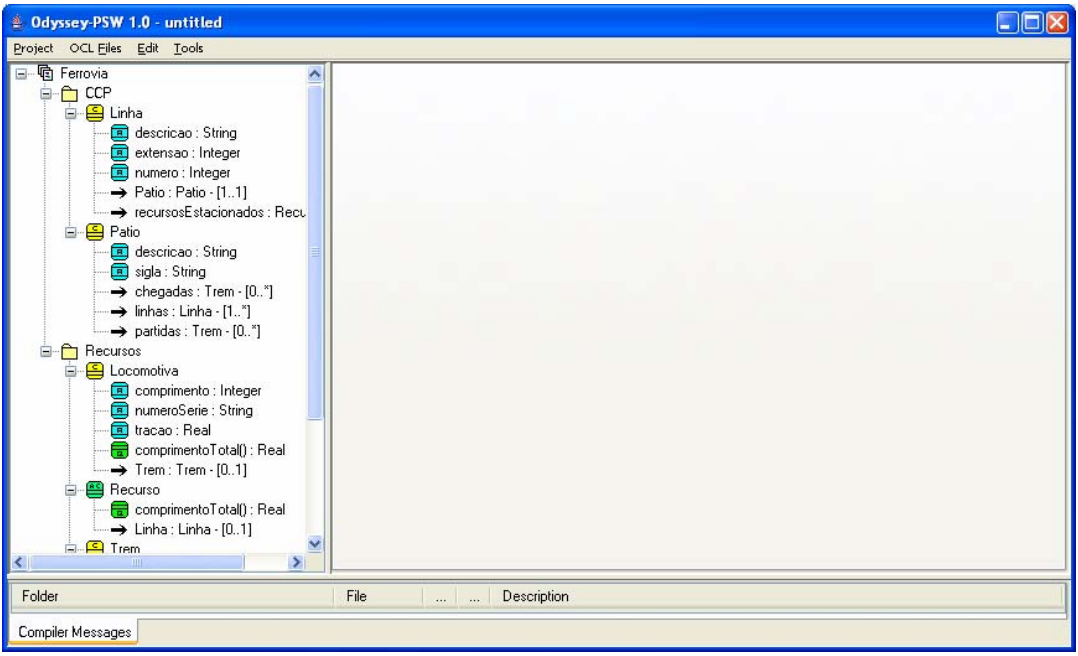


Figura 8.2 – Tela do Odyssey-PSW após importação do modelo

Uma vez que o projeto tenha sido criado, as modificações efetuadas no modelo através de uma ferramenta externa de modelagem podem ser incorporadas ao projeto aberto no Odyssey-PSW por meio de uma opção de atualização *(Project - Refresh)*. A relação dos arquivos que compõem o projeto pode ser modificada através da opção *Project – Properties*, que permite, por exemplo, a inclusão ou a exclusão de arquivos OCL ao projeto.

8.3 Editor e Compilador OCL/OCL-AL

Este módulo oferece recursos de edição e compilação das definições expressas em OCL/OCL-AL referentes aos elementos do modelo associado ao projeto. A estrutura usual para a organização das especificações OCL/OCL-AL de um modelo consiste na elaboração de um arquivo texto para cada classificador que seja utilizado como contexto de definições expressas em OCL/OCL-AL. Dessa forma, cada arquivo contém todas as definições realizadas no contexto do respectivo classificador. Os arquivos são organizados em diretórios de modo a refletir a estrutura de pacotes eventualmente existente no modelo. Embora seja usual, essa estrutura não é obrigatória, ou seja, outras formas de organização podem ser utilizadas com o Odyssey-PSW.

A Figura 8.3 apresenta a janela de edição de especificações OCL/OCL-AL, onde os arquivos são editados em painéis acionados por abas. As seguintes opções de edição

são oferecidas através do menu *OCL Files*: *New*: criar um novo arquivo OCL; *Open*: abrir um arquivo OCL em uma nova aba; *Open All*: abrir, em abas diferentes, todos os arquivos OCL associados ao projeto; *Save* e *Save As*: salvar o conteúdo editado; *Close* e *Close All*: remover a aba de edição do arquivo selecionado ou todas as abas de edição.

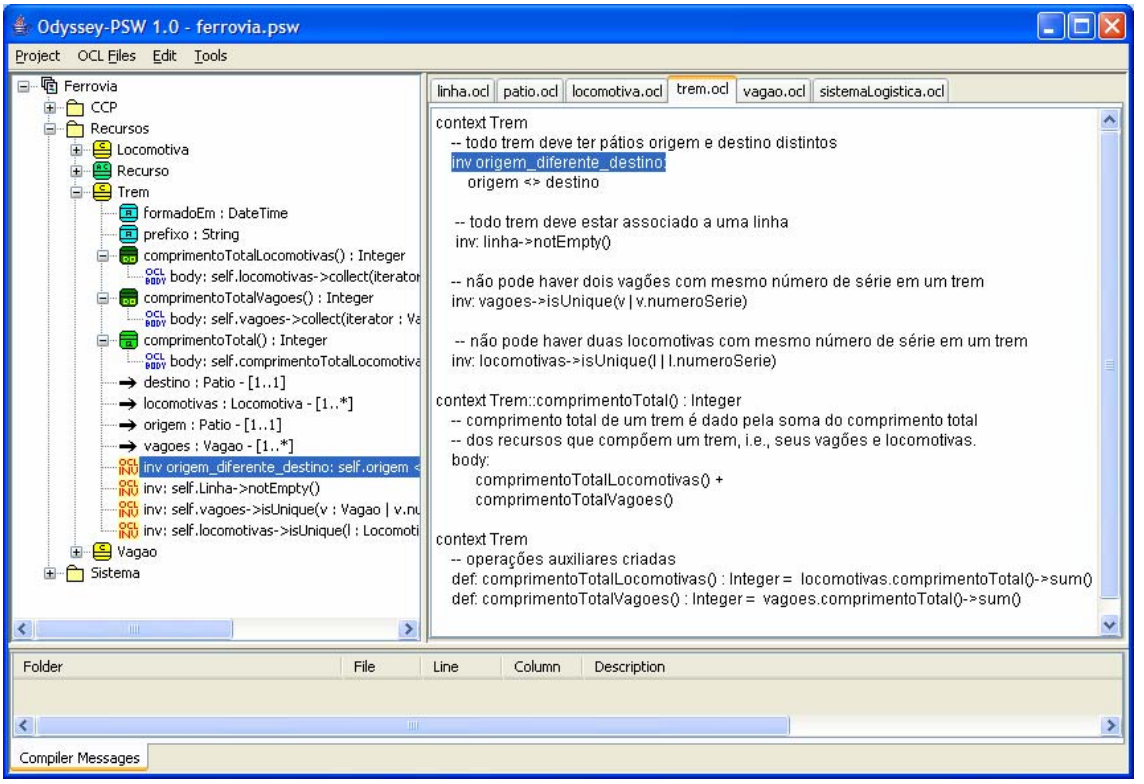


Figura 8.3 – Janela de edição de arquivos OCL/OCL Action Language

O compilador OCL/OCL-AL é compatível com a versão 2.0 da especificação OCL. A análise semântica das definições OCL/OCL-AL de um projeto é realizada em dois passos. No primeiro passo, todas as definições de atributos e operações efetuadas através de cláusulas *def* são acrescentadas aos classificadores do modelo. No segundo passo, todas as demais partes da especificação, incluindo as expressões associadas a essas cláusulas *def*, são analisadas. Dessa forma, as cláusulas *def* podem ser criadas em qualquer ordem e em qualquer arquivo do projeto. No exemplo apresentado na Figura 8.3, a definição da expressão associada à operação de consulta *Trem::comprimentoTotal*() utiliza duas operações, *comprimentoTotalLocomotivas* e *comprimentoTotalVagoes,* que são definidas por cláusulas *def* em um trecho posterior à sua utilização.

Os problemas detectados durante a compilação de um arquivo específico (opção *Tools – Compile OCL File*), ou de todos os arquivos do projeto (opção *Tools – Compile Project*), são apresentados em um painel localizado na parte inferior da janela de edição. A seleção de um erro nesse painel ativa a edição do arquivo no ponto onde o erro foi detectado, conforme ilustrado pela Figura 8.4.

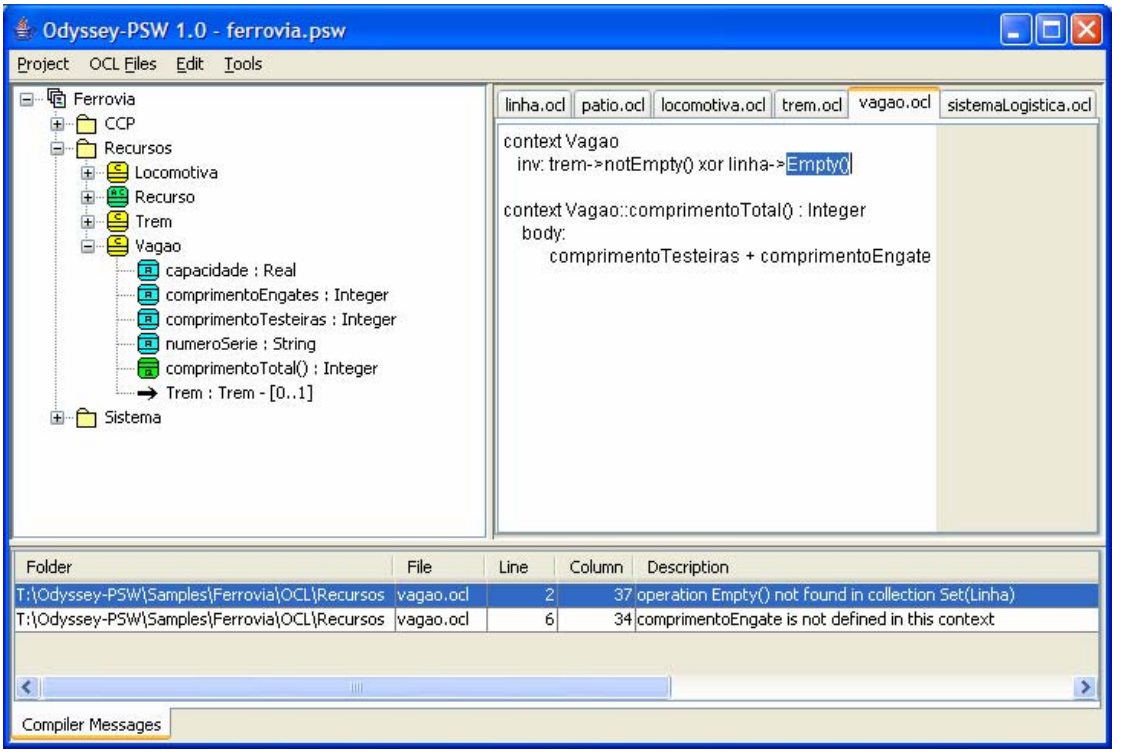


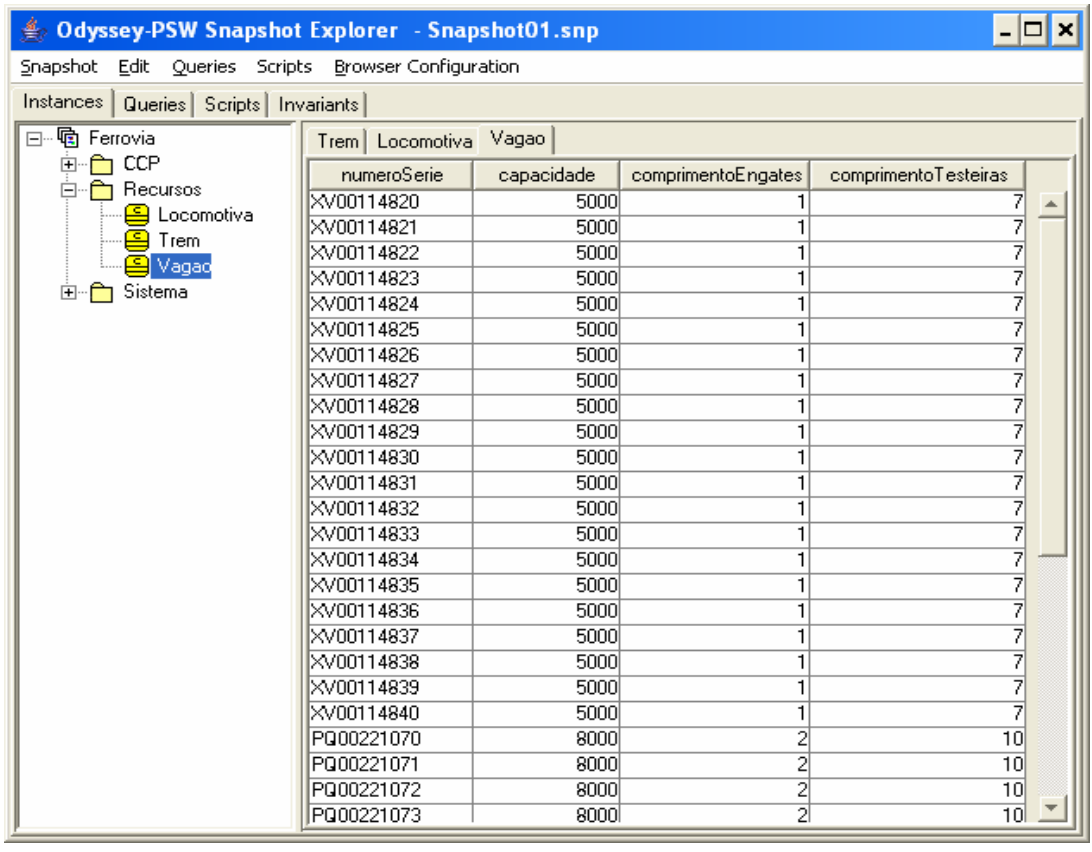
Figura 8.4 – Painel de mensagens de erro do compilador

8.4 Gerenciador de Espaços de Objetos

O gerenciador de espaços de objetos oferece recursos para a criação, manutenção, exploração e avaliação de conjuntos de instâncias das classes de um modelo. Esses recursos têm o propósito de auxiliar o engenheiro de software na tarefa de avaliar se a teoria definida pelo modelo é adequada para a representação de cenários concretos do mundo que foi modelado. Esses conjuntos de instâncias também podem ser utilizados para registrar exemplos concretos do mapeamento dos elementos existentes no mundo que foi modelado para os elementos definidos no modelo, bem como na avaliação de cenários de execução de operações do modelo, conforme será descrito posteriormente neste capítulo.

Todas as operações disponíveis no gerenciador de instâncias, como por exemplo, criação, modificação e exclusão de instâncias e de ligações entre instâncias, consultas, execução de scripts e avaliação de restrições, são realizadas sobre um espaço de objetos. Um espaço de objetos contém todos os objetos e as ligações entre objetos existentes no sistema em um determinado instante, considerando todas as classes e associações do modelo. Múltiplos espaços de objetos podem ser construídos, sendo cada espaço identificado por um nome.

O gerenciador de instâncias é ativado a partir do menu principal através da opção *Tools – Snapshot Explorer*. A Figura 8.5 apresenta a janela principal desse gerenciador. No painel localizado no lado esquerdo dessa janela, uma árvore contendo todas as classes concretas (classes instanciáveis) do modelo é exibida. O painel da direita apresenta todas as instâncias da classe selecionada que existam no espaço de objetos em edição. A barra de título dessa janela apresenta o nome do espaço de objetos em edição no momento que, nesse exemplo, é *Snapsho01*.



numeroSerie	capacidade	comprimentoEngates	comprimentoTesteiras
XV00114820	5000	1	7
XV00114821	5000	1	7
XV00114822	5000	1	7
XV00114823	5000	1	7
XV00114824	5000	1	7
XV00114825	5000	1	7
XV00114826	5000	1	7
XV00114827	5000	1	7
XV00114828	5000	1	7
XV00114829	5000	1	7
XV00114830	5000	1	7
XV00114831	5000	1	7
XV00114832	5000	1	7
XV00114833	5000	1	7
XV00114834	5000	1	7
XV00114835	5000	1	7
XV00114836	5000	1	7
XV00114837	5000	1	7
XV00114838	5000	1	7
XV00114839	5000	1	7
XV00114840	5000	1	7
PQ00221070	8000	2	10
PQ00221071	8000	2	10
PQ00221072	8000	2	10
PQ00221073	8000	2	10

Figura 8.5 – Janela principal do gerenciador de instâncias

8.4.1 Criação, Modificação e Exclusão de Instâncias

A criação de instâncias das classes de um modelo pode ser realizada interativamente por meio de uma interface gráfica. Ao selecionar-se a opção *Edit – Create Instance* a partir da janela principal do gerenciador de instâncias, uma janela de edição específica para a classe em exibição no painel direito é apresentada. A Figura 8.6 apresenta a janela para criação/edição de uma instância da classe *Trem* do modelo exemplo.

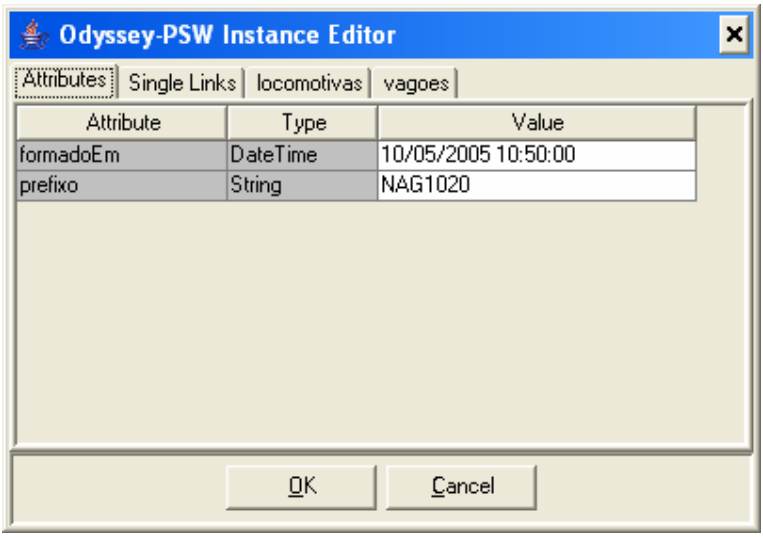


Figura 8.6 - Janela de edição das propriedades de uma instância

A edição das propriedades de uma instância é dividida em um conjunto de abas. Os valores para os atributos de uma instância podem ser definidos através da aba *Attributes*. Atributos derivados, ou adicionados ao modelo através da cláusula *def,* também são exibidos nessa aba. Entretanto, seus valores não podem ser editados, pois são atualizados automaticamente pelo software.

O painel apresentado na aba *SingleLinks* permite a definição de ligações entre a instância que está sendo criada/editada e instâncias de classes associadas com multiplicidade máxima de 1. A Figura 8.7 apresenta um exemplo desse painel para um cenário de criação de uma instância da classe *Trem*. Através desse painel, é possível criar ou excluir ligações entre a instância de *Trem* e seus pátios origem e destino, bem como entre o trem e a linha onde ele porventura esteja estacionado. Essas ligações correspondem às associações da classe *Trem* com as classes *Pátio* e *Linha* definidas no modelo apresentado na Figura 8.1.

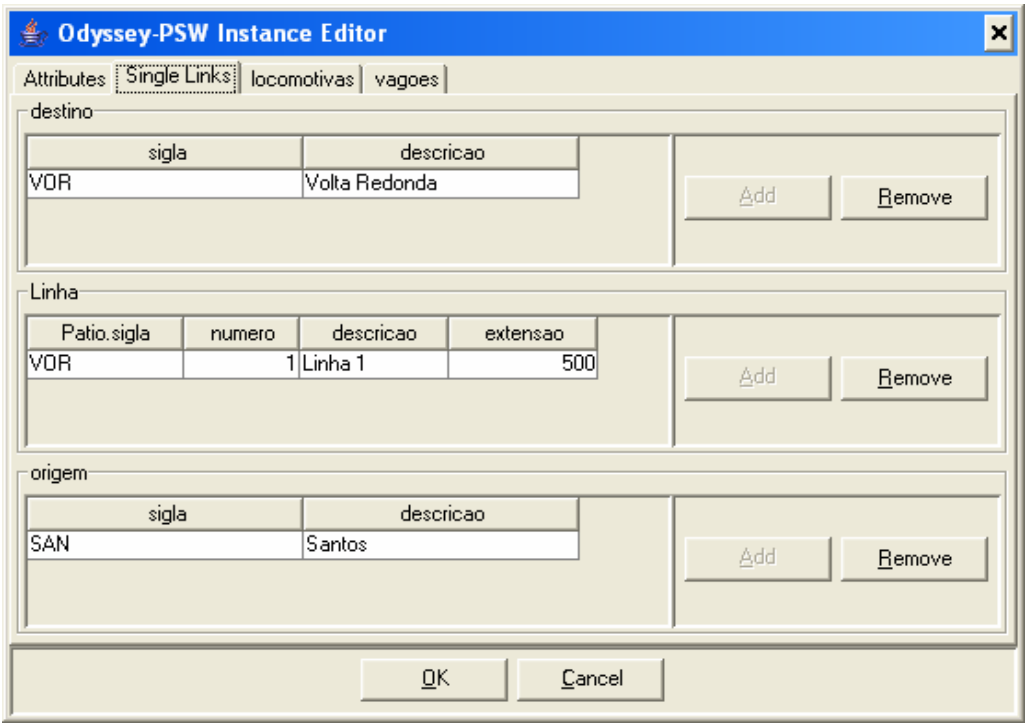


Figura 8.7 - Janela de edição de ligações entre instâncias (multiplicidade 1)

A criação ou exclusão de ligações com instâncias de classes associadas com multiplicidade maior que 1 é efetuada através de painéis definidos em abas específicas para cada associação. Como a multiplicidade das classes *Locomotiva* e *Vagão* nas respectivas associações com a classe *Trem* é maior que 1, as ligações de uma instância de trem com instâncias dessas classes são criadas ou removidas em abas específicas (uma aba para as locomotivas e outra para os vagões), conforme ilustrado na Figura 8.8.

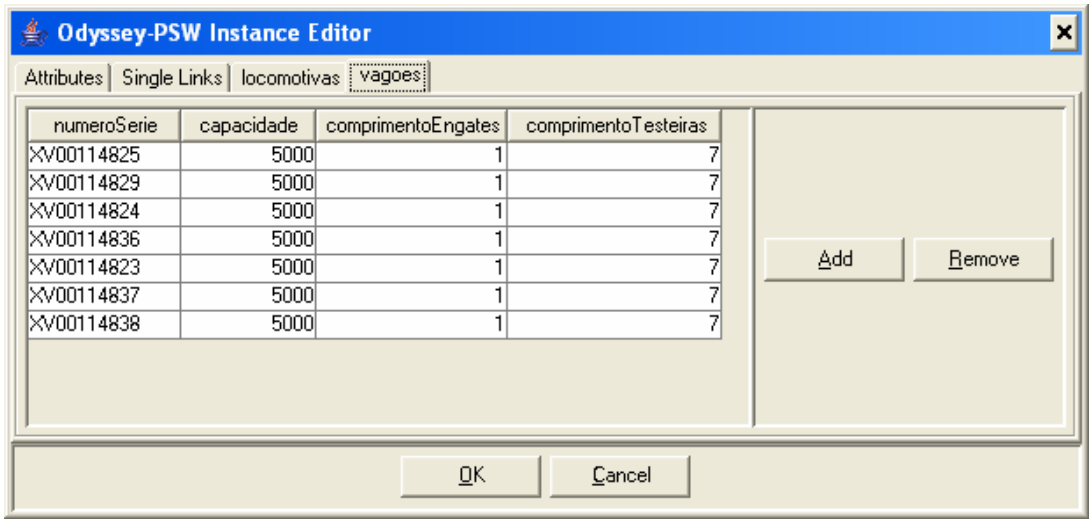


Figura 8.8 - Janela de edição de ligações entre instâncias (multiplicidade maior que 1)

A modificação dos valores dos atributos de uma instância ou de suas ligações com outras instâncias é efetuada de forma análoga à criação de uma nova instância, bastando selecionar o objeto a ser modificado, e escolher a opção *Edit – Modify Instance* do menu. Outra operação também disponível no menu de edição é a exclusão da instância selecionada, ativada através da opção *Edit – Delete Instance*.

8.4.2 Execução de Scripts

A criação, modificação ou exclusão em lote de instâncias pode ser efetuada através de scripts elaborados em OCL-AL. A Figura 8.9 apresenta a janela de edição e execução de scripts. Os scripts são organizados em uma árvore hierárquica de grupos que é apresentada na aba Scripts, localizada no painel esquerdo. Cada script é identificado por um nome e consiste em um comando ou bloco de comandos OCL-AL que, ao ser executado, poderá gerar modificações no espaço de instâncias que estiver em edição. O script apresentado no exemplo da Figura 8.9 corresponde à criação em lote de 21 instâncias da classe *Vagao*, enquanto que o script presente na Figura 8.10 corresponde à modificação em lote da capacidade e do comprimento de engates de todas as instâncias de vagão cujo número de série seja iniciado por 'PQ'.

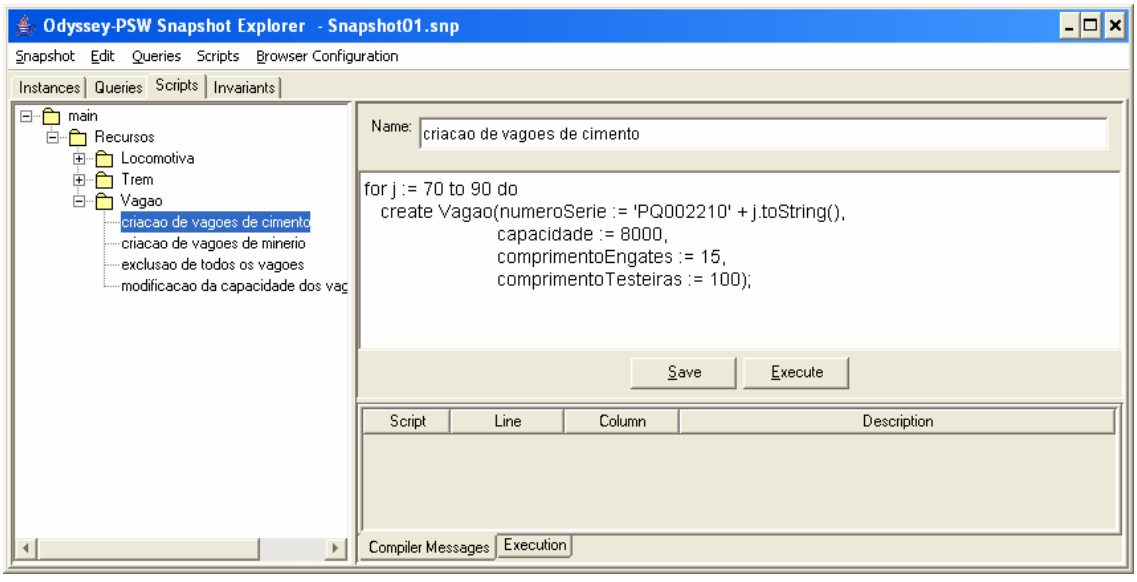


Figura 8.9 - Script para criação em lote de instâncias

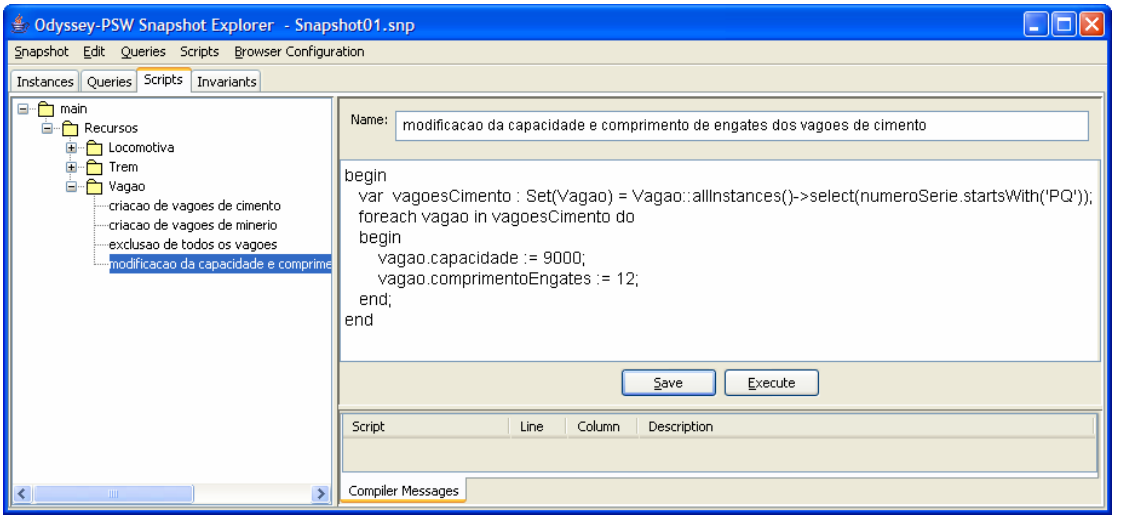


Figura 8.10 – Script para modificação em lote de instâncias

Essa forma de definição de scripts é útil para a realização de pequenas modificações no espaço de instâncias. Para modificações mais complexas, uma forma mais conveniente de criar e executar os scripts consiste em defini-los no contexto de classes especialmente criadas para tal. Nesse caso, o primeiro passo consiste em definir classes cujas operações hospedarão os scripts que manipularão as instâncias do modelo. A Figura 8.11 apresenta um diagrama com a definição de classes que exemplificam algumas possibilidades de organização e reutilização de scripts.

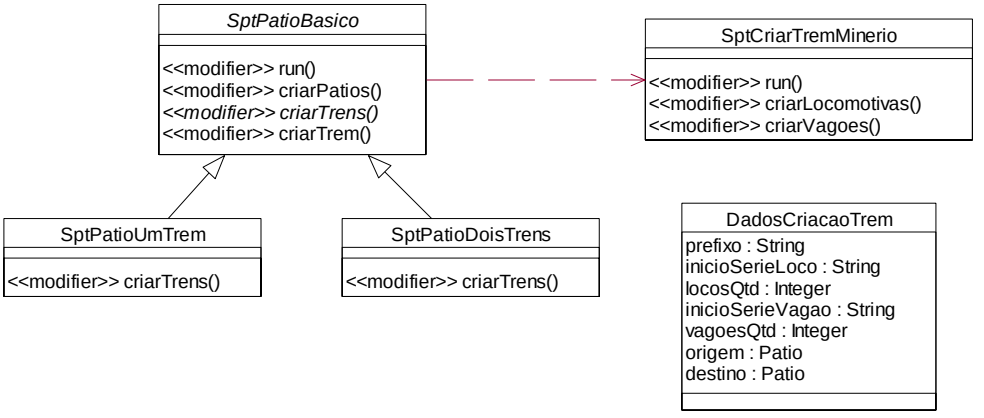


Figura 8.11 - Definição das classes com scripts de modificação de um espaço de instâncias

Nesse diagrama, foram definidas duas classes, *SptPatioUmTrem* e *SptPatioDoisTrens*, que correspondem a scripts de criação de pátios e trens. Esses dois scripts possuem vários trechos em comum e, portanto, foram definidos como subclasses de *SptPatioBasico*, uma classe abstrata que define elementos comuns aos dois scripts.

Além disso, essas duas classes reutilizam um script básico para a criação de um trem de minério, definido na classe *SptCriarTremMinerio*.

Essas classes são definidas como qualquer outra classe do modelo, ou seja, através de uma ferramenta CASE externa de modelagem. Após importar para o Odyssey-PSW o arquivo XMI contendo a definição dessas classes de script, o próximo passo consiste em definir as ações associadas a cada uma das operações definidas nessas classes.

A Figura 8.12 apresenta o script de criação de um trem de minério. Cada operação da classe *SptCriarTremMinerio* tem suas ações especificadas em OCL-AL através da construção *actionBody*. As operações *criarLocomotivas* e *criarVagoes* são chamadas pela operação *run*, que corresponde à ação principal desse script. Dessa forma, a complexidade de um script pode ser gerenciada através da sua divisão em operações menos complexas.

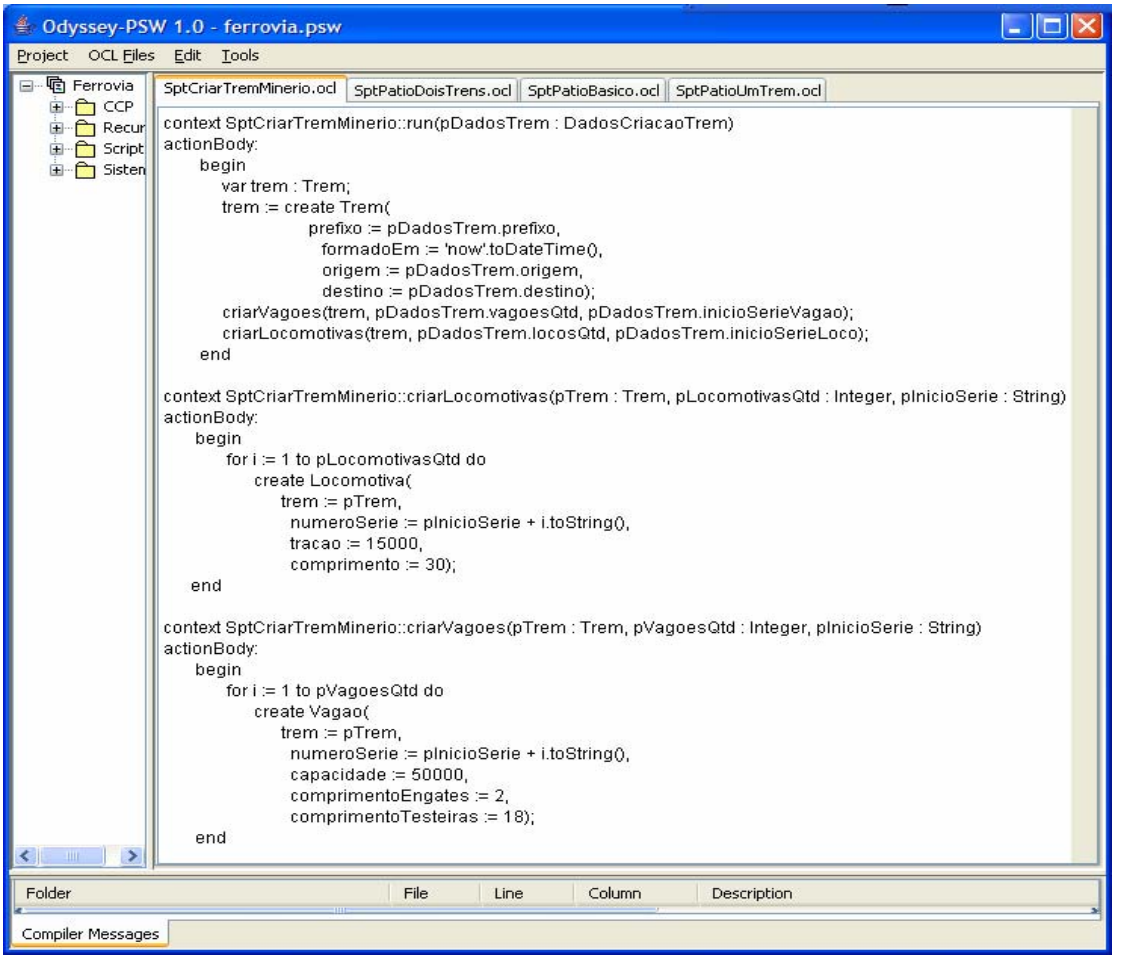


Figura 8.12 - Script de criação de um trem de minério

A Figura 8.13 apresenta a parte comum aos dois scripts de criação de pátios e trens presente definida na classe *SptPatioBasico*. Nesse exemplo, a operação *run* é definida através de uma chamada à operação *criarTrens,* definida apenas nas subclasses (scripts específicos). Além disso, a operação *criarTrem* faz uso do script de criação de um trem de minério apresentado na Figura 8.12.

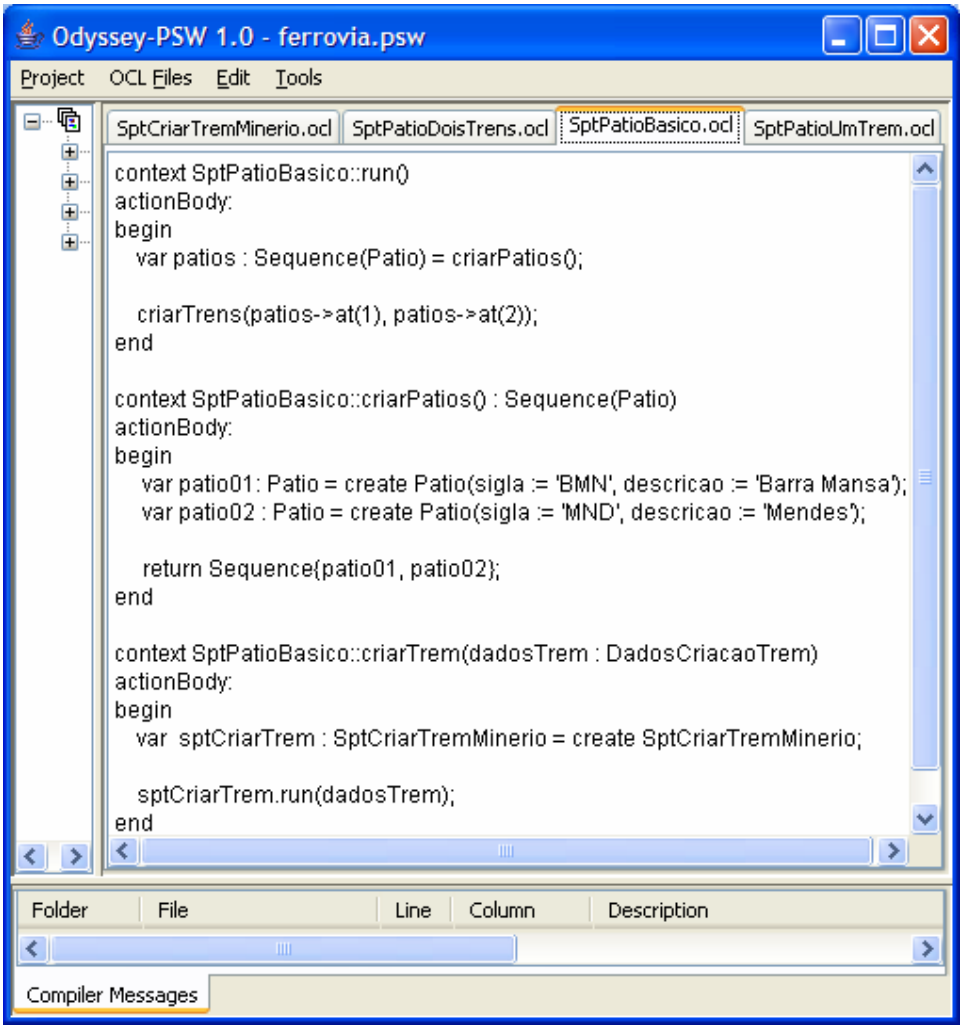


Figura 8.13 - Classe base para scripts específicos

A Figura 8.14 apresenta a definição da operação *criarTrens* no script *SptPatioUmTrem*. Essa operação utiliza a operação *criarTrem* definida na superclasse *SptPatioBasico*. Assim, os recursos oferecidos pelo paradigma orientado a objetos podem ser utilizados na definição dos scripts de forma a evitar redundância, promover a reutilização e uma melhor gestão da complexidade dos mesmos.

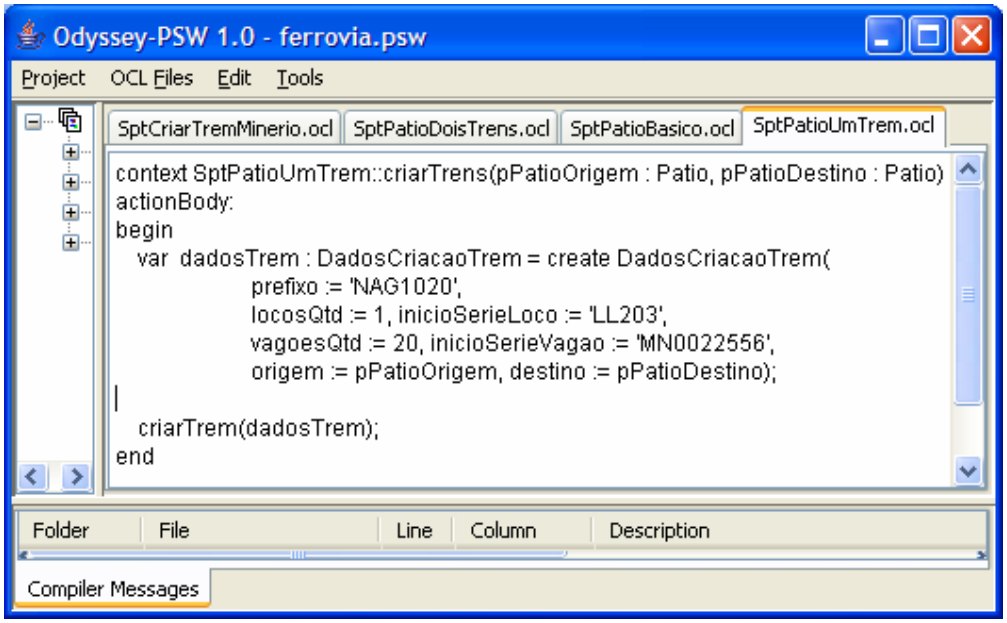


Figura 8.14 - Script associado à subclasse SptPatioUmTrem

A Figura 8.15 mostra como um script definido através de classes pode ser executado. Basta instanciar a classe contendo o script a ser executado (nesse exemplo, *SptPatioUmTrem*), e definir uma chamada à operação correspondente ao ponto de entrada desse script, que, nesse exemplo, é a operação *run*.

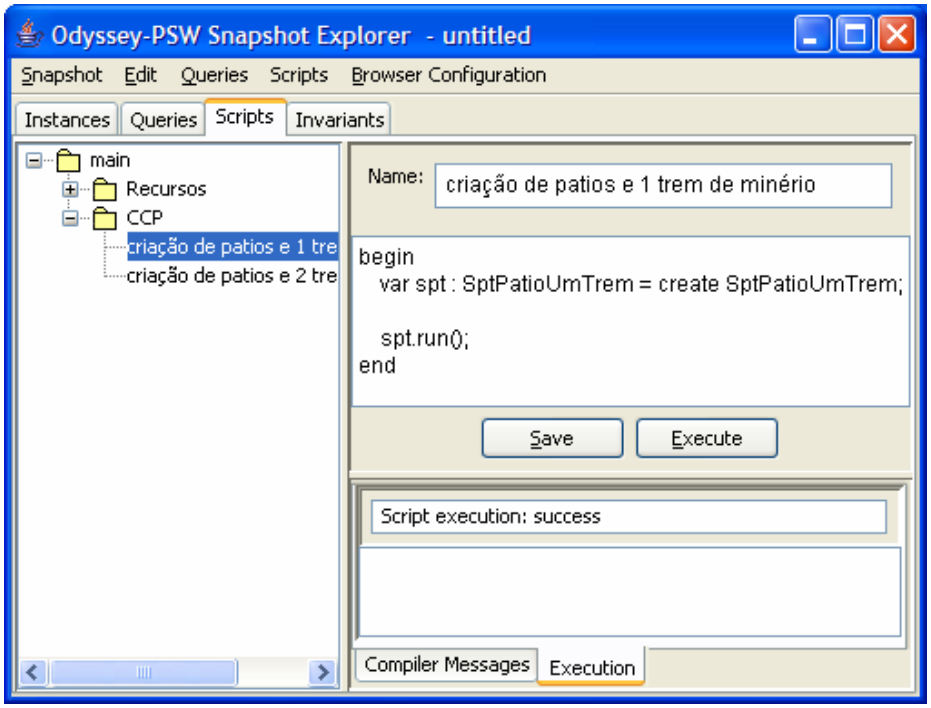


Figura 8.15 - Execução de um script definido em uma classe específica para essa finalidade

Como os scripts são definidos na forma de operações fornecidas por classes de um modelo, eles podem ser especificados através de precondições e pós-condições associadas a essas operações. O exemplo da Figura 8.16 ilustra como a operação *criarTrens* da classe de script *SptPatioUmTrem* pode ser especificada em termos de precondições e pós-condições. Esta especificação contém uma pós-condição incorreta (*novosTrens->size() = 2),* que indica que duas novas instâncias de trem devem ser criadas em função da execução dessa operação.

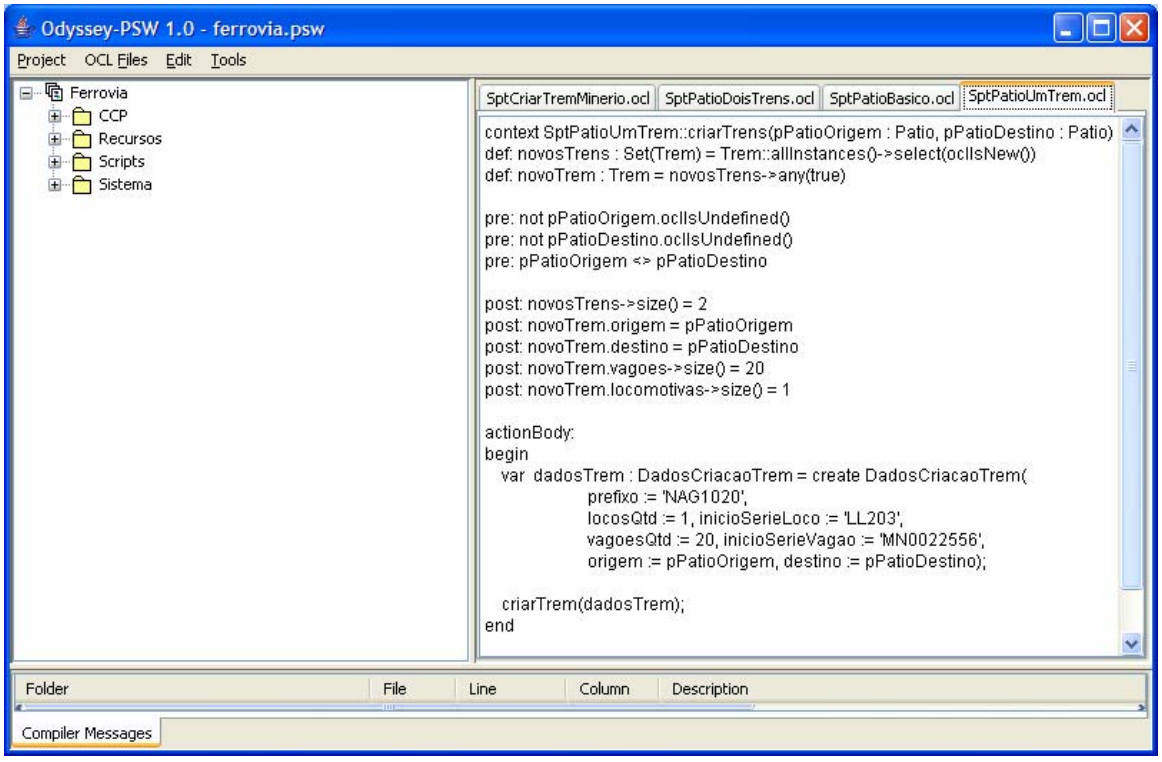


Figura 8.16 - Exemplo de especificação de pré e pós-condições para um script

A Figura 8.17 ilustra como a violação dessa pós-condição é apresentada na execução desse script. O painel inferior direito apresenta a pós-condição violada e a sua respectiva avaliação que, nesse exemplo, indica que apenas uma instância de trem foi criada pela execução da operação, e não duas instâncias, como definido na pós-condição.

Um script específico pode ser executado através do botão *Execute* no painel de edição do mesmo e atua sobre o espaço de instâncias em edição. Para executar todos os scripts de um grupo, basta selecioná-lo no painel esquerdo e ativar a opção *Execute All* do menu associado à árvore de grupos. Os scripts podem ser carregados e armazenados de forma independente do espaço de instâncias em edição através de operações

disponíveis no menu *Scripts (Load, Save)*, o que torna possível sua execução sobre diferentes espaços de instâncias.

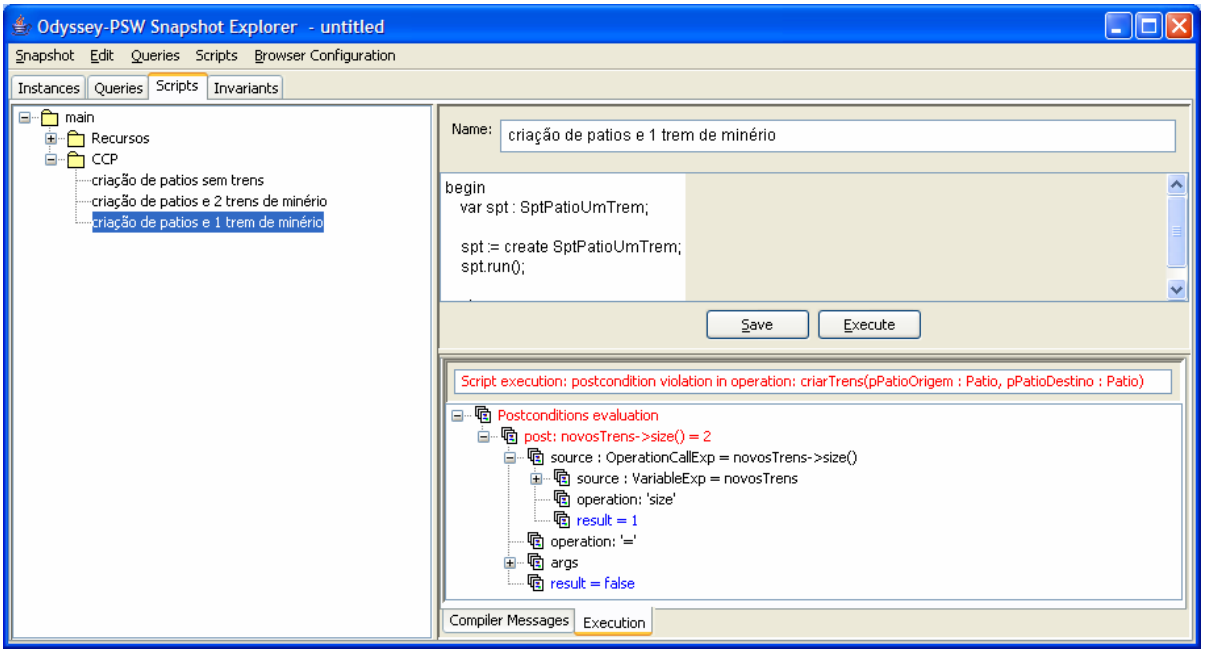


Figura 8.17 - Exemplo de violação da especificação de um script

8.4.3 Consultas ad-hoc

Este módulo permite que consultas ad-hoc sejam definidas e executadas sobre o espaço de instâncias em edição. Através dessas consultas, é possível visualizar as instâncias, seus atributos e relacionamentos de formas diferentes daquela disponível na exibição padrão das instâncias apresentada na Figura 8.5. Filtros, ordenações específicas e tuplas resultantes da extração de informações de instâncias de diferentes classes do modelo são alguns exemplos do que é possível obter através dessas consultas.

Cada consulta tem um nome que a identifica e uma expressão OCL que será avaliada sobre o espaço de instâncias. É possível executar uma consulta específica através do botão *Execute* no painel de edição da mesma, ou ainda executar todas as consultas de um grupo, selecionando o grupo no painel esquerdo e ativando a opção *Evaluate All* do menu. De forma análoga aos scripts, as consultas podem ser armazenadas e avaliadas em outros espaços de instâncias.

A Figura 8.18 apresenta a tela de edição e execução de consultas ad-hoc. As consultas podem ser organizadas em uma estrutura hierárquica de grupos que é apresentada no painel esquerdo. A expressão OCL correspondente à consulta é

especificada no painel direito, e o seu resultado é apresentado em três diferentes visões (*Table View*, *Tree View* e *AST View*). O exemplo da Figura 8.18 apresenta uma consulta que resulta em todas as instâncias da classe *Vagão* que contenham um valor iniciado por 'PQ' para o atributo *numeroSerie*, onde as instâncias resultantes são ordenadas por numeroSerie. Nesse exemplo, o resultado da consulta está apresentado na forma tabular (aba *Table View*).

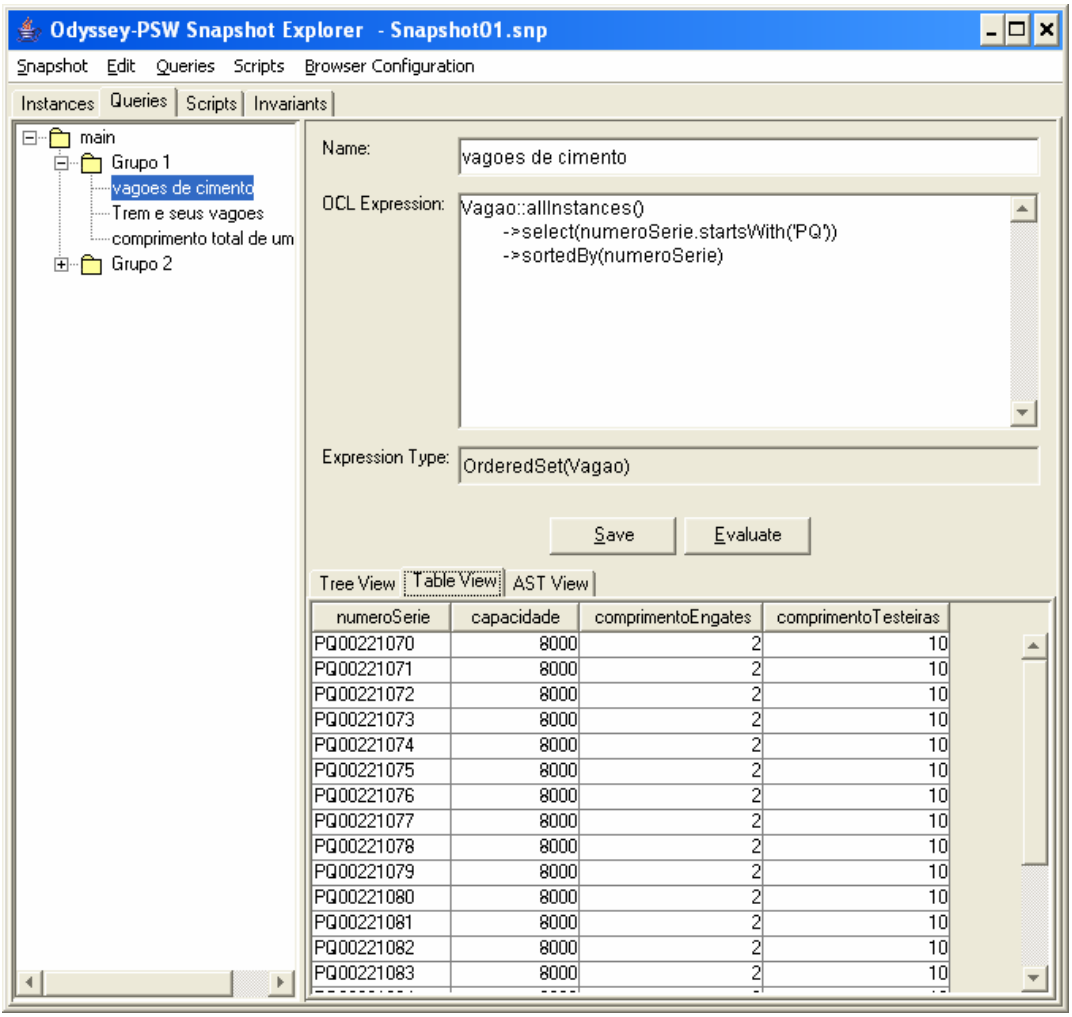


Figura 8.18 - Tela de consulta ad-hoc

A Figura 8.19 apresenta um exemplo de uma consulta que coleta informações de instâncias de mais de uma classe, gerando uma coleção de tuplas como resultado. Nesse exemplo, a coleção de tuplas resultante é obtida extraindo-se de cada trem o seu prefixo, o número de série, o comprimento total de cada um de seus vagões e o somatório do comprimento total dos vagões que o compõem. O resultado é exibido em uma árvore onde os elementos podem ser expandidos ou comprimidos de acordo com a

conveniência do usuário. No exemplo da Figura 8.19, o resultado corresponde a um trem de prefixo NAG1020 que é composto por 6 vagões, sendo que o primeiro tem o número de série XV00114825 e o comprimento total de 80, e o comprimento total dos 6 vagões é 560. Essa forma de visualização (*Tree View)* é útil para consultas que resultem em estruturas aninhadas de informações através da utilização de tuplas.

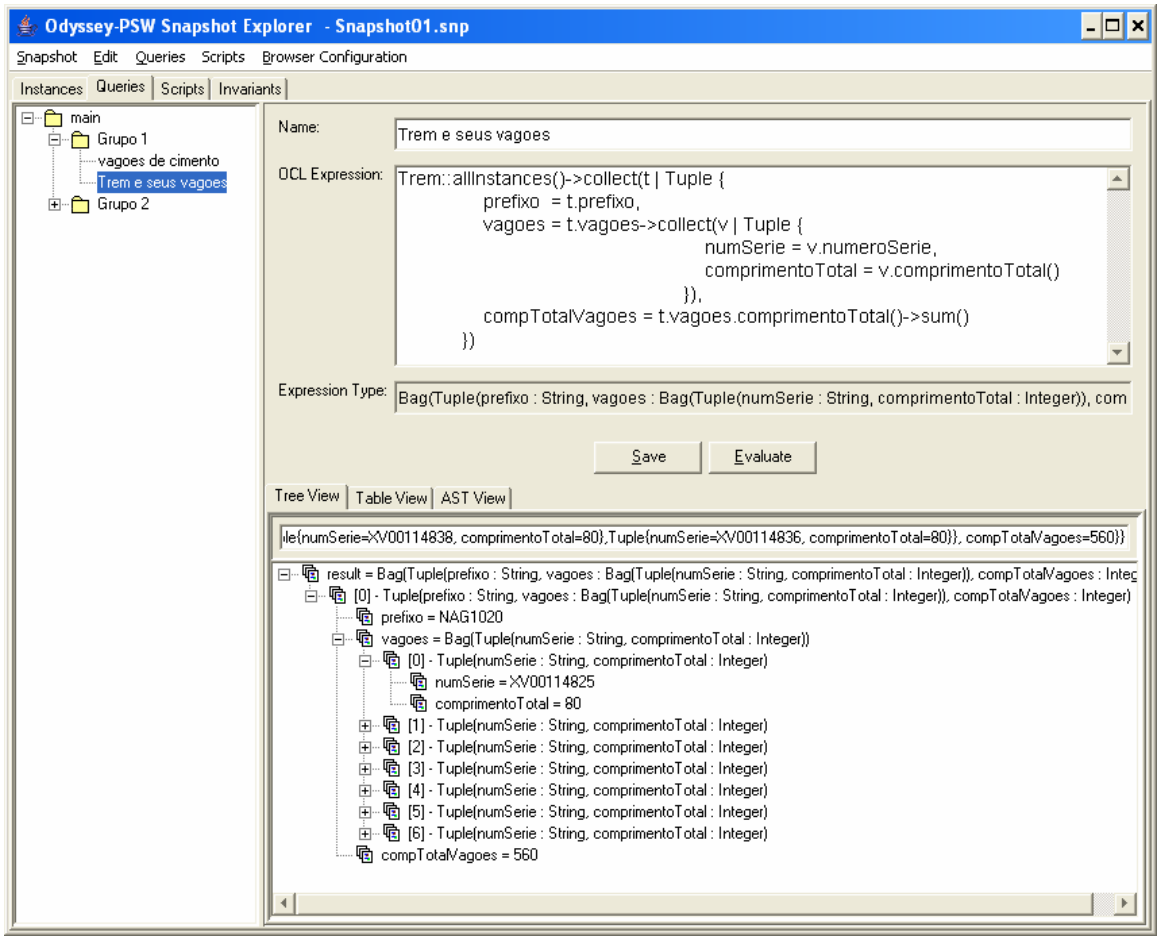


Figura 8.19 - Exemplo de uma consulta visualizada na forma de árvore de resultados

Os resultados de uma consulta também podem ser exibidos em uma terceira forma de visualização, denominada *AST View*. Nessa visão, uma expressão OCL é apresentada na forma de instâncias do metamodelo OCL, ou seja, no formato da árvore semântica abstrata resultante do processo de compilação da expressão. O resultado da avaliação de cada nó da árvore é apresentado, permitindo a visualização das avaliações parciais dos elementos que compõem a expressão. Assim, essa forma de apresentação do resultado de uma consulta tem como objetivo possibilitar que o usuário possa

entender como as expressões são avaliadas, bem como identificar a razão da obtenção de resultados eventualmente não esperados.

A Figura 8.20 apresenta um exemplo de visualização do resultado de uma consulta na forma *AST View*. Nesse exemplo, a expressão avaliada é a operação de consulta *comprimentoTotal* aplicada ao vagão de número de série XV00114820. A operação *comprimentoTotal* está definida na classe *Vagão* e corresponde à expressão *comprimentoTesteiras* + *comprimentoEngates*. O resultado é apresentado na forma de árvore, onde o resultado da avaliação de cada nó é apresentado em uma cor diferenciada dos nós referentes às instâncias de classes do metamodelo OCL como, por exemplo, *OperationCallExp*, *VariableExp*, entre outras.

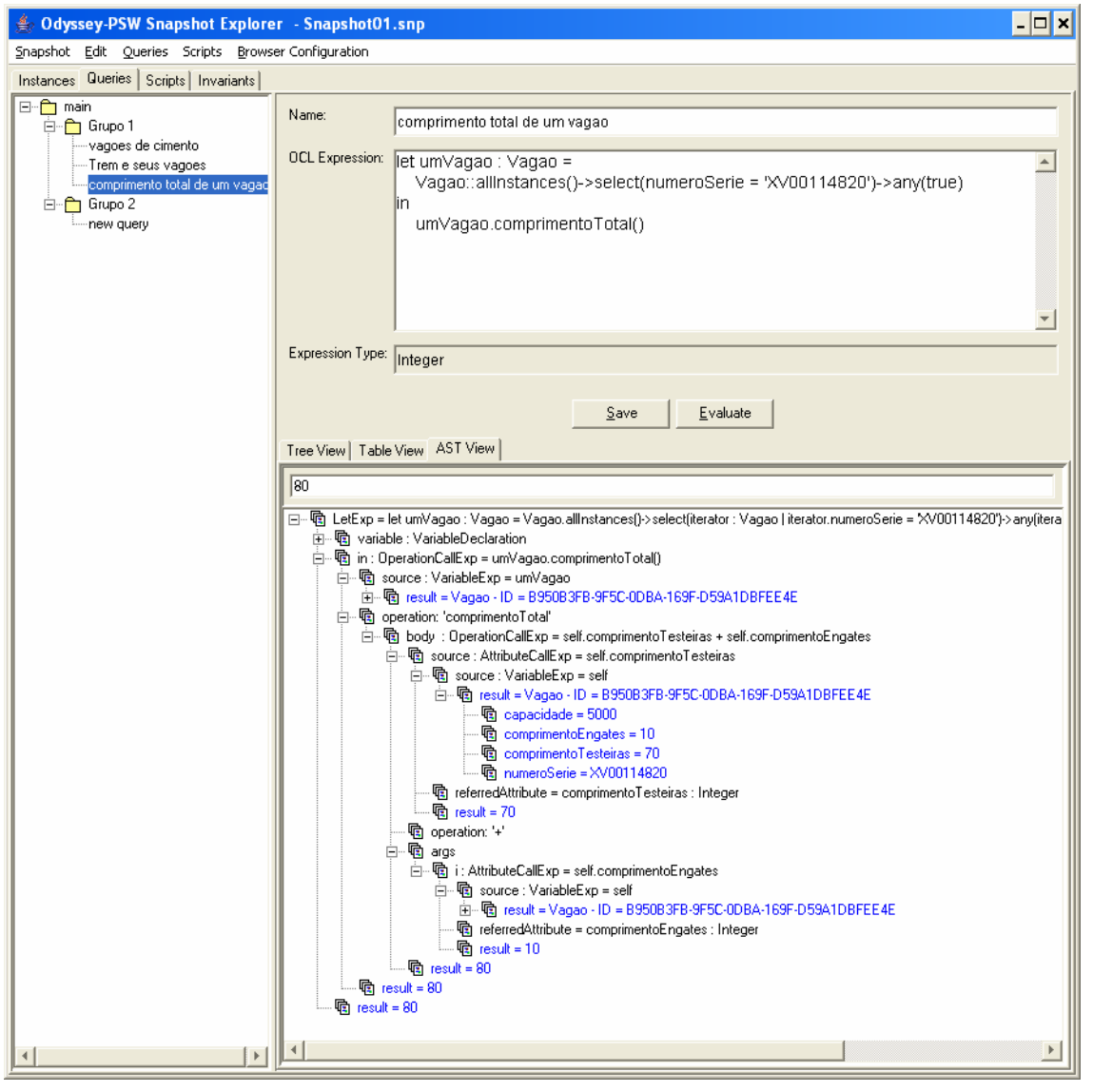


Figura 8.20 - Exemplo de uma consulta visualizada na forma *AST View*

O diagrama de objetos da Figura 8.21 apresenta a mesma expressão de consulta - *umVagao.comprimentoTotal()* - na forma de instâncias do meta-modelo da OCL. Os nós da árvore apresentada na Figura 8.20 correspondem aos objetos desse diagrama.

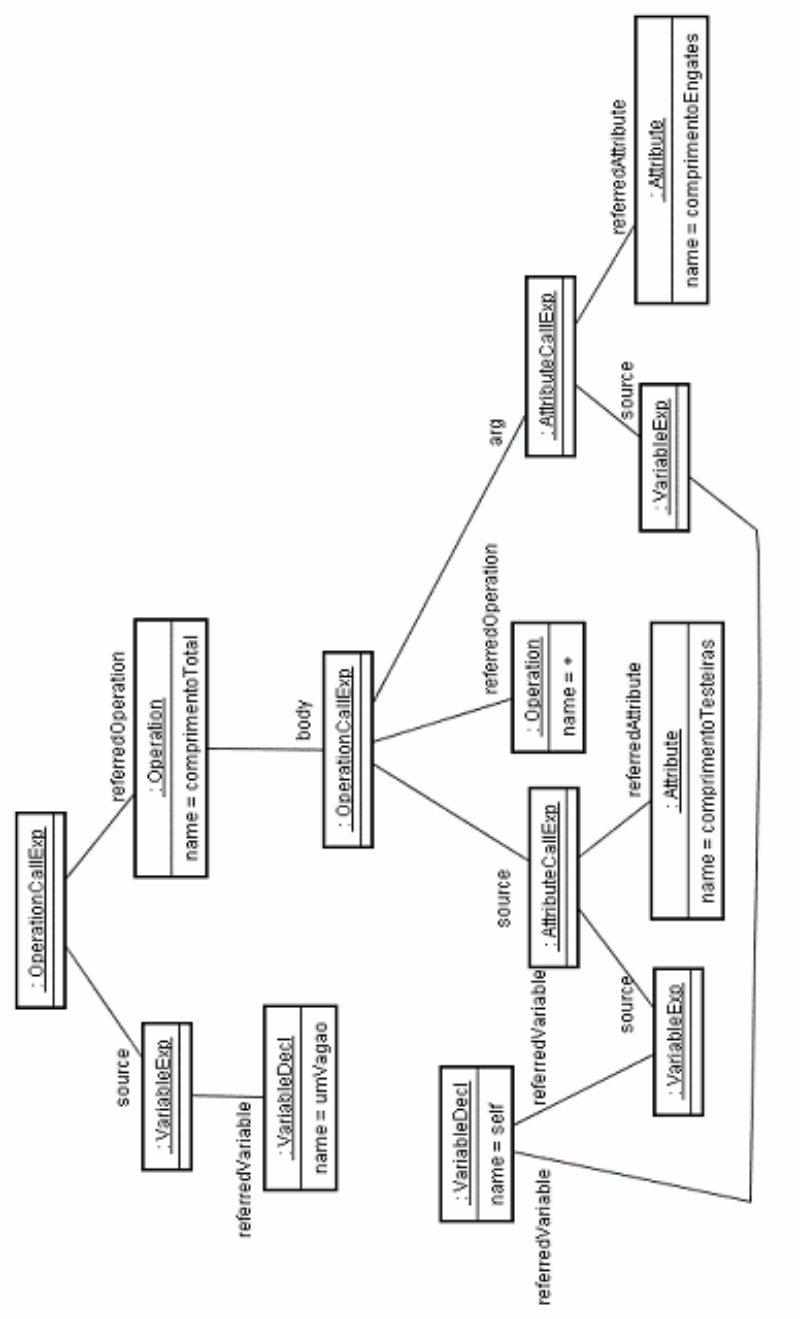


Figura 8.21 - Diagrama de objetos correspondente à compilação da expressão do exemplo

8.4.4 Avaliação de Invariantes e Multiplicidades

Um modelo pode definir diversas restrições que devem ser respeitadas pelas suas instâncias. O modelo elaborado em uma ferramenta externa de modelagem, e importado pelo Odyssey-PSW, define restrições de multiplicidade para as ligações entre as instâncias das classes desse modelo. Outras restrições podem ser adicionadas ao modelo através de invariantes especificadas em OCL. Através deste módulo de avaliação de invariantes e multiplicidades, é possível verificar se todas essas restrições estão sendo respeitadas por um espaço de instâncias. A violação de alguma dessas restrições pode indicar problemas na definição no modelo ou na definição do espaço de instâncias.

A Figura 8.22 apresenta a janela de avaliação de restrições. Ela é acessada através da aba "*Invariants*" disponível no editor de instâncias. O painel esquerdo apresenta as restrições associadas a cada classe do modelo. Cada classe está associada a dois grupos de restrições: *OCL Invariants*, que contém todas as restrições expressas em OCL definidas no contexto dessa classe, e *Multiplicity Constraints*, que contém todas as restrições ligadas à multiplicidade das associações que envolvam essa classe. Apenas as multiplicidades que tenham um valor mínimo diferente de 0 ou um valor máximo diferente de * são avaliadas, uma vez que a multiplicidade *0..** não estabelece nenhuma restrição às ligações entre as instâncias das classes envolvidas.

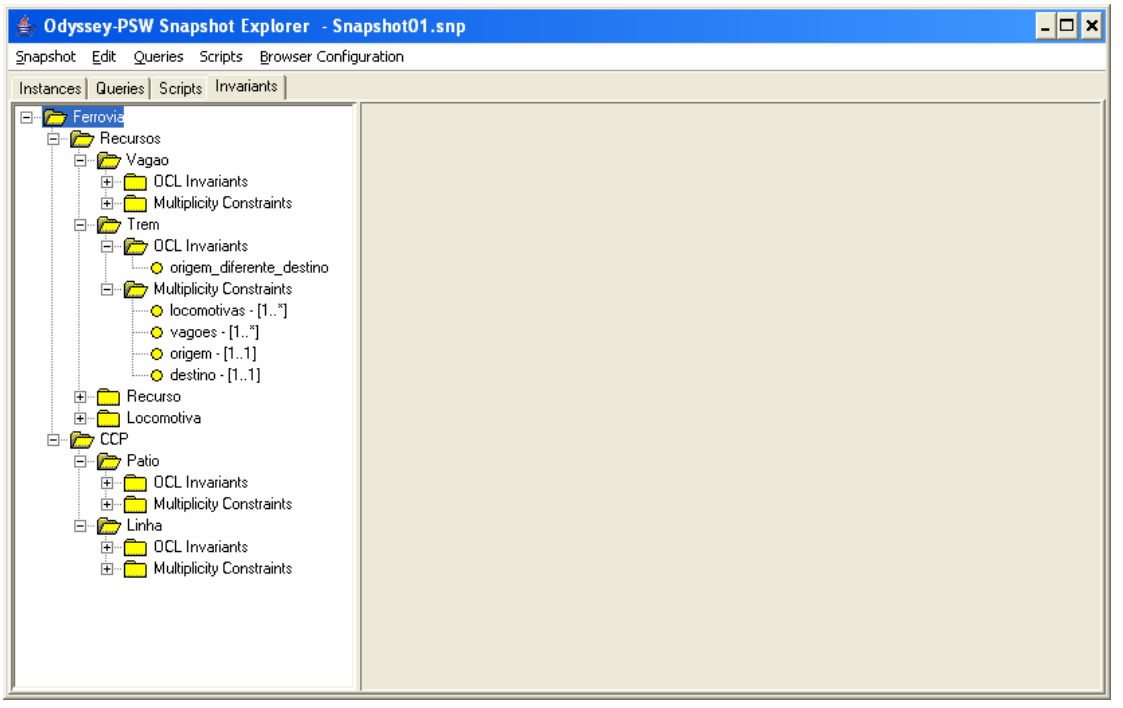


Figura 8.22 - Tela de avaliação de invariantes e multiplicidades

O usuário pode solicitar a avaliação de uma restrição específica ou de um grupo de restrições selecionando o elemento a ser avaliado (restrição, classe, pacote ou modelo) na árvore localizada no painel esquerdo, e acionando a opção *Evaluate*. A Figura 8.23 apresenta a janela resultante da avaliação de todas as restrições do modelo. Todas as restrições que não tiverem sido violadas são indicadas pela cor verde, enquanto que todas as restrições violadas são indicadas pela cor vermelha.

No caso de violação de uma restrição de multiplicidade, o painel direito indica a multiplicidade esperada e uma relação das instâncias que a violaram. No exemplo da Figura 8.23, o trem de prefixo NAG1020 não tem nenhuma locomotiva associada, enquanto que a associação entre *Trem* e *Locomotiva* determina que todo trem deva estar associado, ao menos, a uma locomotiva.

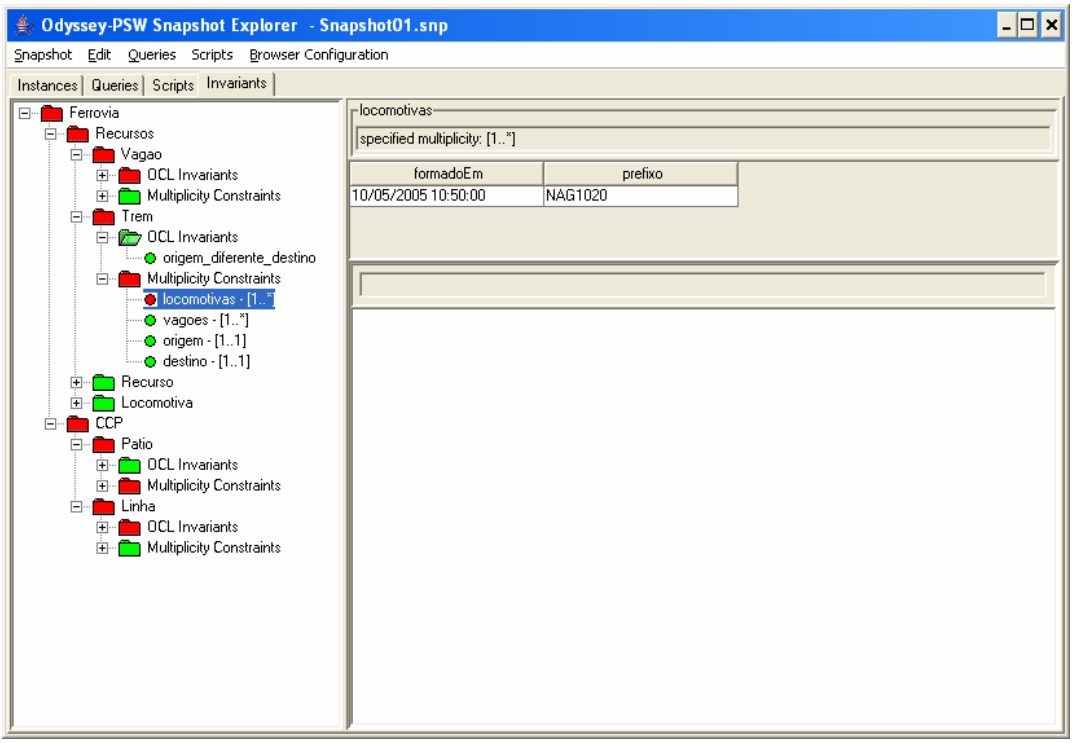


Figura 8.23 - Avaliação de restrições de multiplicidade

No caso de restrições expressas em OCL, o painel direito apresenta a relação das instâncias que violaram uma determinada restrição e o detalhamento da avaliação da expressão OCL no formato *AST View* descrito anteriormente para a instância selecionada. No exemplo ilustrado pela Figura 8.24, existe uma instância de linha que está violando a restrição que especifica que o valor do atributo extensão deve ser sempre maior ou igual ao valor da ocupação total da linha *(self.extensao* >=

self.ocupacaoTotal()). No painel inferior, pode-se observar que o resultado da operação *ocupacaoTotal* da linha (*result = 560*) é superior ao valor da extensão da linha (*result = 500*), indicando que, se a restrição estiver corretamente especificada, esse espaço de instâncias não é válido.

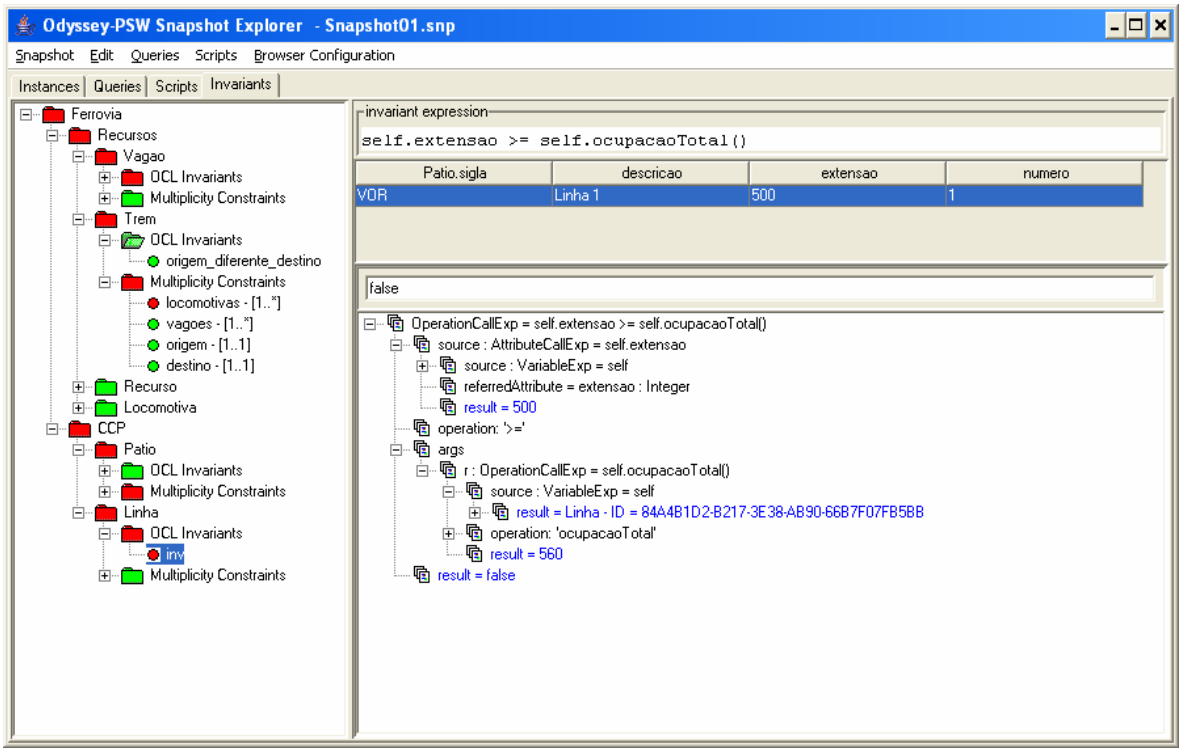


Figura 8.24 - Avaliação de invariantes definidas em OCL

8.4.5 Configuração da Visualização Tabular de Instâncias

Este módulo permite definir, para cada classe do modelo, as informações que serão apresentadas na visualização tabular de um conjunto de suas instâncias. Alguns exemplos de situações onde essa forma de visualização é utilizada podem ser vistos na Figura 8.5, na Figura 8.8 e na Figura 8.18. A Figura 8.25 apresenta a janela onde essa definição é realizada. Após escolher a classe a ser configurada no painel esquerdo, o usuário define, a partir dos itens disponíveis para essa classe, aqueles que serão exibidos (através dos botões *Add* e *Remove*) e a sua ordem de exibição (através dos botões *Up* e *Down*). O conjunto de itens disponíveis para exibição é formado pelos atributos da classe, considerando o fechamento transitivo de todas as suas superclasses, os atributos das classes diretamente associadas e os identificadores (GUID) da instância da classe e de todas as instâncias associadas.

No exemplo da Figura 8.25, a exibição de instâncias da classe *Linha* conterá todos os seus atributos (*número*, *descrição* e *extensão*), precedidos pelo atributo *sigla* da instância da classe *Pátio* associada. Como dois pátios podem ter linhas com o mesmo número, a exibição da sigla do pátio permite identificar uma determinada linha mais facilmente, como pode ser visto na Figura 8.26, que apresenta uma visão tabular de instâncias da classe Linha segundo a configuração efetuada.

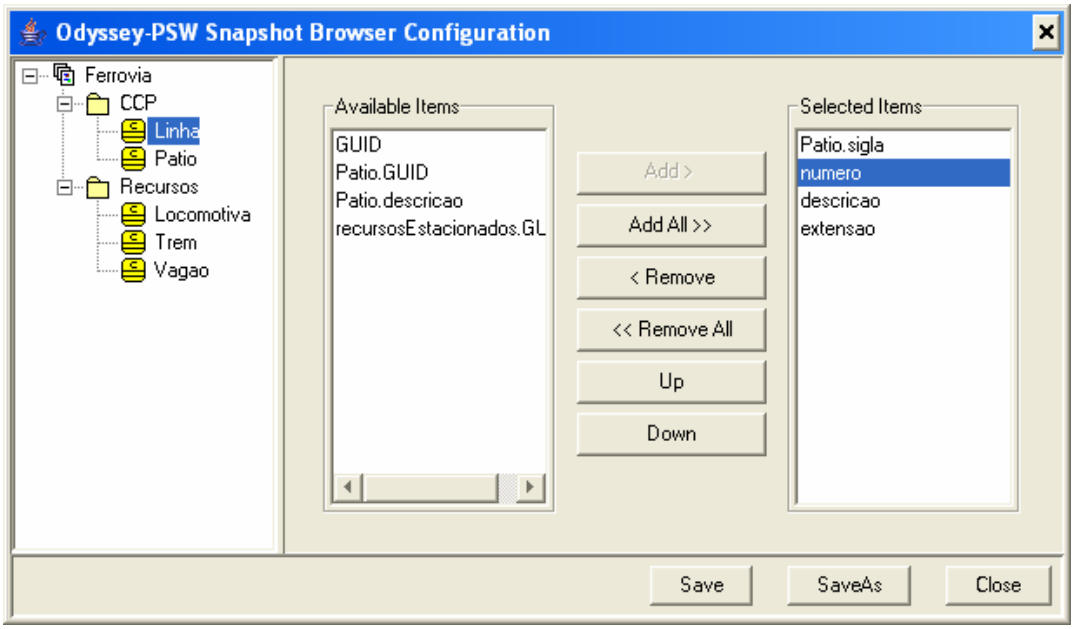


Figura 8.25 - Configuração da visualização tabular de instâncias

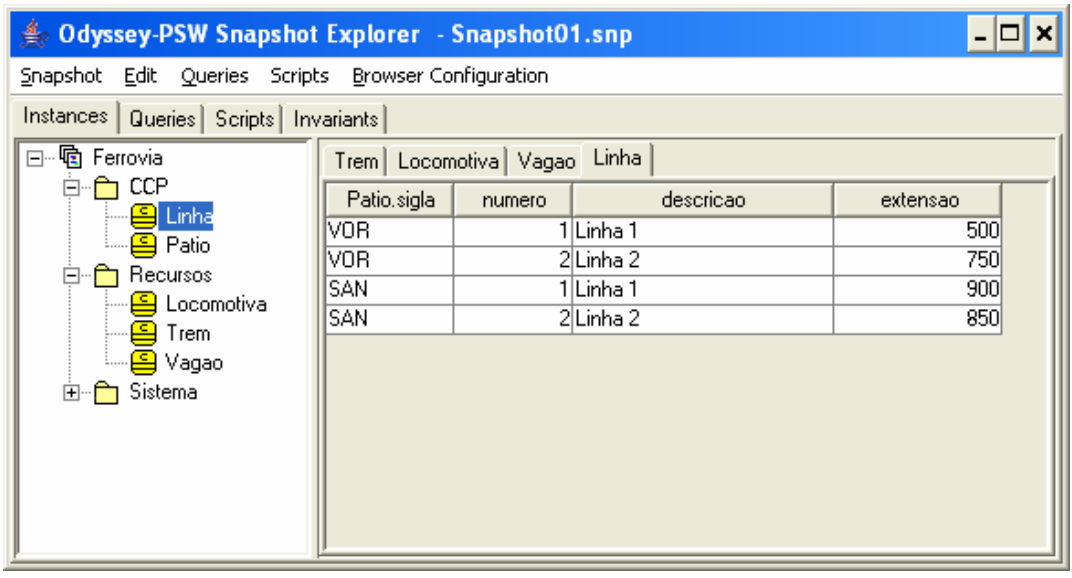


Figura 8.26 - Exemplo do resultado da configuração da visão tabular das instâncias

8.5 Avaliação de Expressões de Consulta

Este módulo oferece recursos para a criação, manutenção e execução de casos de testes para as expressões OCL que definem o corpo das operações de consulta definidas no modelo, das operações adicionadas ao modelo através da cláusula *def* e também das expressões OCL associadas a atributos derivados, tendo como objetivo apoiar a validação destas expressões.

Cada cenário é definido por um nome, uma expressão OCL que será avaliada sobre um espaço de instâncias previamente configurado através do módulo gerenciador de instâncias descrito anteriormente, e uma expressão OCL correspondente ao resultado esperado desta avaliação. Os casos de teste são organizados em uma hierarquia de grupos, sendo que, para cada grupo, é possível definir o espaço de instâncias que será utilizado na avaliação de todos os casos de teste definidos no respectivo grupo.

A Figura 8.27 apresenta a janela de edição de um grupo. Nesse exemplo, o grupo *Recursos* foi associado ao espaço de instâncias *Snapshot01*. Dessa forma, todos os casos de teste definidos nesse grupo serão avaliados sobre o espaço de instâncias *Snapshot01*. Esse mesmo espaço de instâncias será utilizado na avaliação dos casos de teste definidos nos grupos direta ou indiretamente subordinados ao grupo *Recursos* para os quais não seja definido um espaço de instâncias específico.

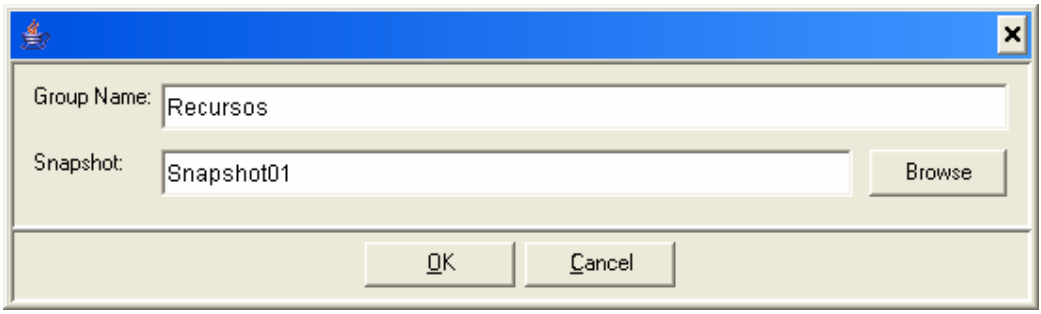


Figura 8.27 - Janela de definição de um grupo de casos de teste

A Figura 8.28 apresenta a janela onde os casos de teste são criados e avaliados. O painel esquerdo apresenta uma estrutura hierárquica com os grupos e casos de teste definidos pelo usuário. Nesse exemplo, foram criados quatro casos de teste para a operação *comprimentoTotal* definida na classe *Trem* e dois casos de teste para a operação *comprimentoTotal* definida na classe *Vagão*. O cenário apresentado no painel

direito corresponde à avaliação do comprimento total do trem com prefixo QVT4050, composto por uma locomotiva e um vagão, onde o valor esperado como resultado é 23.

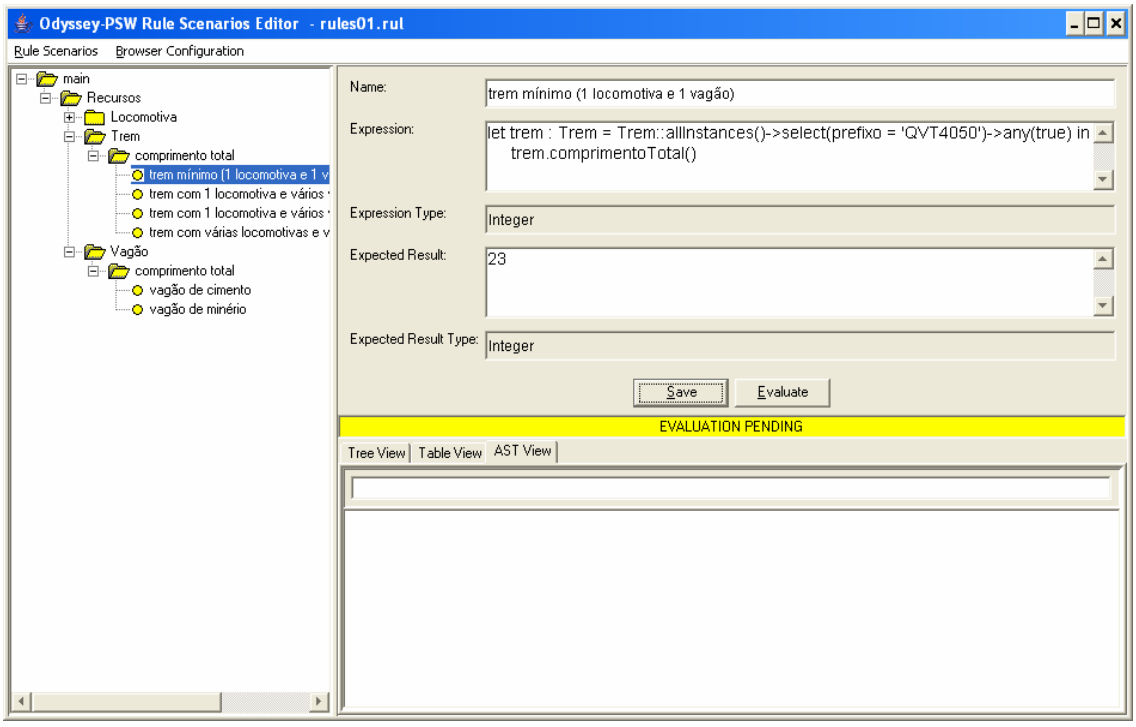


Figura 8.28 - Tela do avaliador de expressões de consulta

É possível executar a avaliação de um cenário específico através do botão *Execute* no painel de edição, ou ainda, executar todos os casos de teste de um grupo, selecionando o grupo no painel esquerdo e ativando a opção *Evaluate All* no menu. A Figura 8.29 apresenta a tela resultante da avaliação de casos de teste de operações de consulta. Todos os casos de teste cujas avaliações resultem nos valores esperados são indicados pela cor verde, enquanto que os casos de teste cujas avaliações resultem em valores diferentes daqueles esperados são indicados pela cor vermelha. O resultado de cada avaliação pode ser visualizado nas três formas apresentadas na seção 6.4.3 (*Tree View, Table View e AST View)*. No exemplo apresentado na Figura 8.29, a avaliação da expressão é apresentada na forma *AST View* que permite ver que o resultado 23 foi obtido pela soma de *comprimentoTotalLocomotivas* (result = 15) e *comprimentoTotalVagoes* (result = 8).

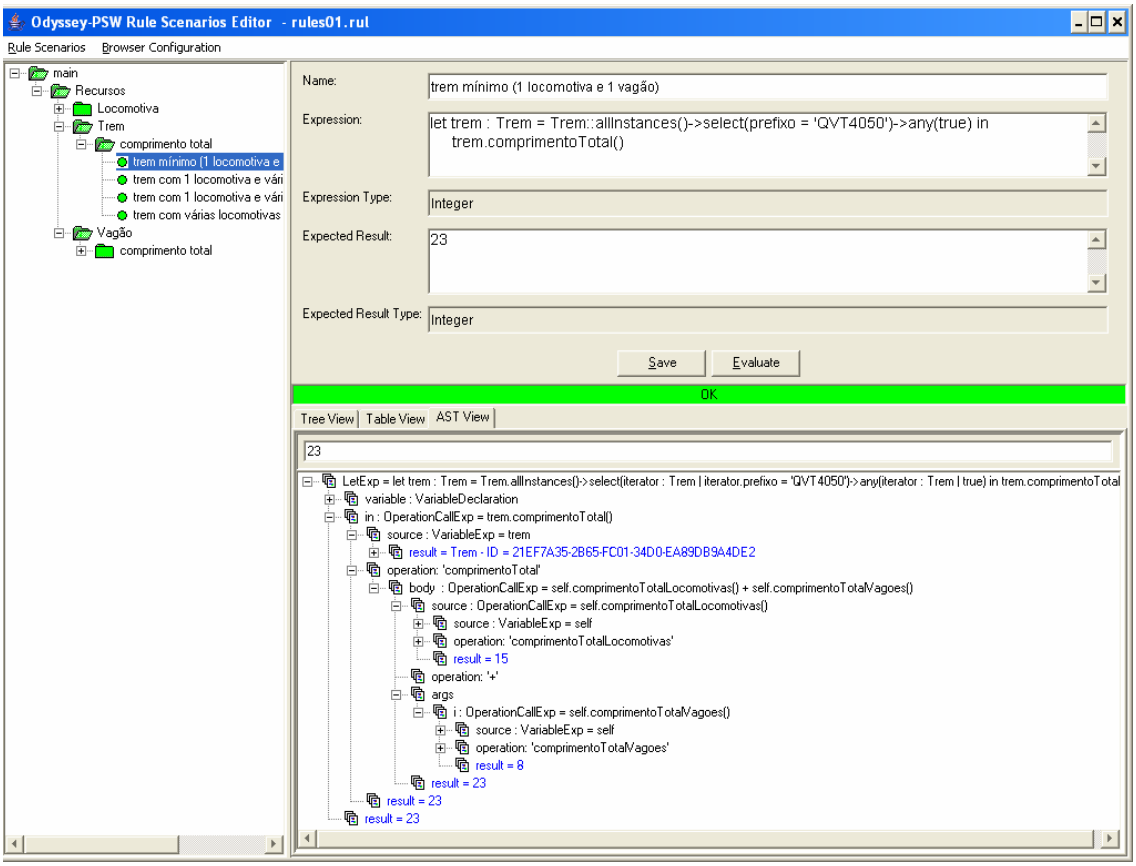


Figura 8.29 – Tela de avaliação de expressões de consulta após a execução

8.6 Avaliação de Especificações Implícitas

Este módulo oferece recursos para a criação, manutenção e execução de casos de teste para especificações implícitas (pré e pós-condições) de operações do modelo. A avaliação destes casos de teste pode ser realizada por meio de execução simulada ou pela execução da especificação explícita da operação. A especificação explícita de operações de consulta é definida por uma expressão OCL através da construção *body*, enquanto que a especificação explícita de operações modificadoras é definida por um conjunto de ações através da cláusula *actionBody*, tornando possível a sua execução. A Figura 8.30 apresenta a especificação da operação modificadora *moverRecurso* que move um recurso para uma outra linha do pátio (*linhaDestino*).

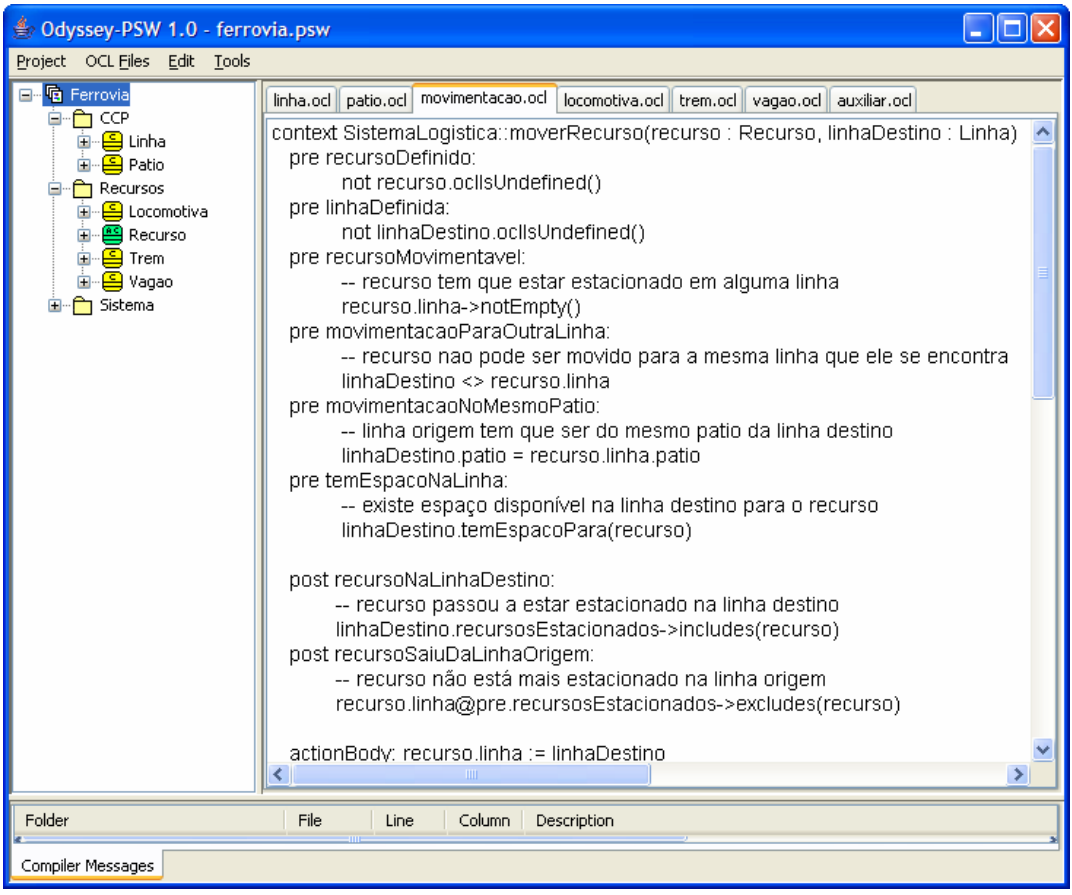


Figura 8.30 - Exemplo de especficação de uma operação modificadora

A avaliação das pré e pós-condições de uma operação pode ser realizada através de casos de teste. Cada caso de teste é definido por um nome, uma configuração inicial de objetos e uma seqüência de chamadas de operações. Um cenário pode corresponder a uma invocação específica de uma operação ou a uma seqüência de invocações da mesma operação ou de operações distintas. Os casos de teste são organizados em uma hierarquia de grupos.

A Figura 8.31 apresenta a janela principal deste módulo. A hierarquia de casos de teste definida pelo usuário é apresentada na árvore localizada no painel esquerdo. Os grupos são representados visualmente por pastas e os casos de teste por círculos. O painel da direita apresenta o cenário selecionado na árvore. Nesse exemplo, o cenário de nome "Locomotiva para linha quase lotada" corresponde à execução da operação *moverRecurso*, que moverá a locomotiva LL2030 para a linha 5 do pátio VOR. O espaço de objetos imediatamente anterior à execução desse cenário, definido pelo campo *Initial Snapshot,* será *Snapshot02*, que deve ser construído através do

gerenciador de espaços de objetos descrito anteriormente. Através dos botões *Add*, *Edit* e *Remove*, invocações de operações do modelo podem ser adicionadas, editadas ou removidas. Os botões *Up* e *Down* permitem modificar a ordem de execução das invocações de operações que compõem o cenário.

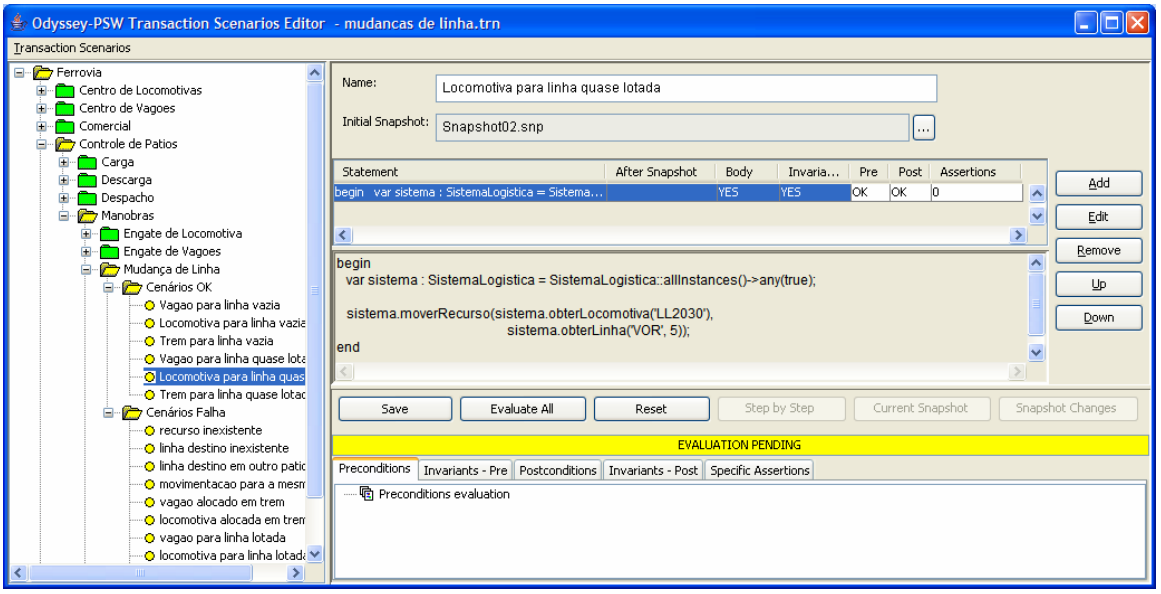


Figura 8.31 - Hierarquia de casos de teste de avaliação de operações modificadoras

Uma invocação de operação é editada através da janela ilustrada pela Figura 8.32. No campo S*tatement*, pode ser definida uma chamada simples de operação ou um bloco de comandos OCL-AL finalizado por uma chamada de operação. Os resultados da avaliação das pré e pós-condições da operação esperados para a chamada especificada são definidos nos campos *Expected Preconditions Evaluation* e *Expected PostconditionsEvaluation* (*OK ou FAIL*). O campo *Evaluate Invariants* indica se as restrições globais do modelo, especificadas através de invariantes, devem ser avaliadas em conjunto com as pré e pós-condições.

Existem duas formas possíveis para a avaliação das pós-condições de cada chamada de operação do cenário. A forma a ser utilizada é indicada pela opção *Use Action Body when available*:

- opção *Use Action Body when available = YES:* a avaliação é efetuada sobre um espaço de instâncias gerado pela execução da especificação explícita da

operação. Nesse caso, deve existir uma especificação explícita para a operação.

- opção *Use Action Body when available = NO*: a avaliação é efetuada sobre um espaço de instâncias pré-configurado através do gerenciador de espaços de objetos, que corresponde ao instante imediatamente posterior à execução da operação. Esse espaço de instâncias é indicado no campo *Post Snapshot.*

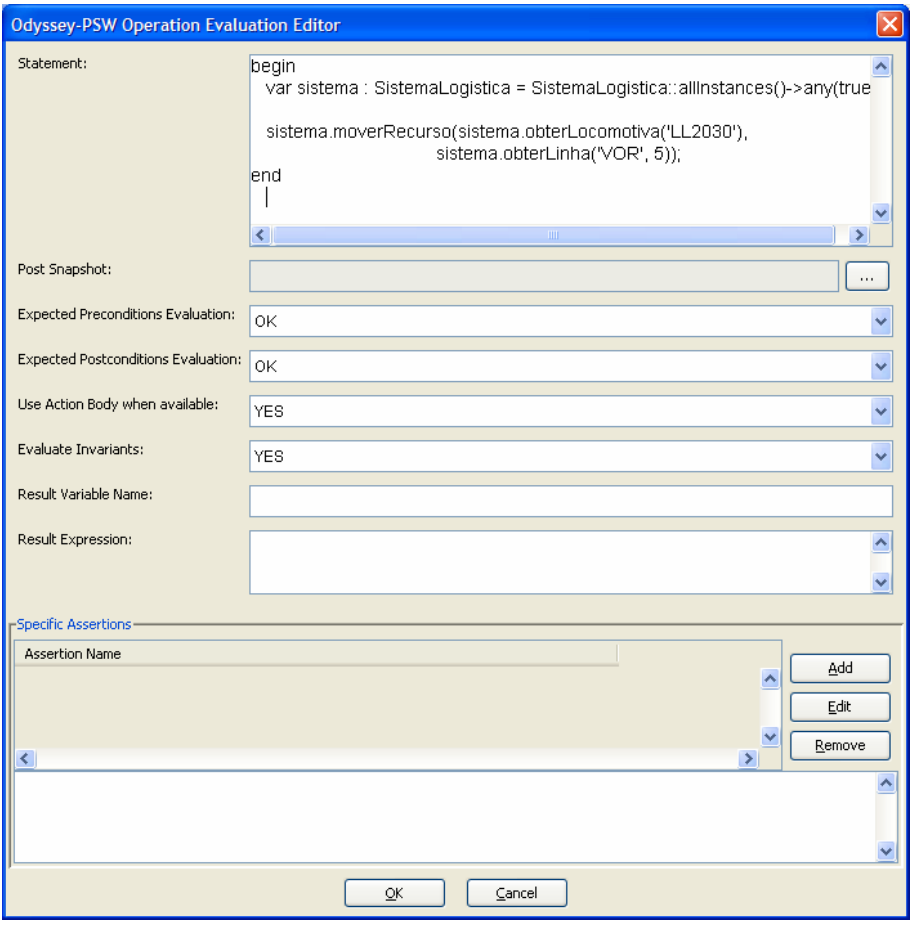


Figura 8.32 - Janela de edição de uma chamada de operação que faz parte de um cenário

A Figura 8.33 apresenta um exemplo de definição de um cenário composto por uma seqüência de chamadas a operações. Esse cenário consiste em:

a) mover o trem NAG1020 para a linha 3 do pátio VOR;

b) mover o vagão QT105566 para a linha 2 desse pátio;

c) formar um trem na linha 2 com a locomotiva LL2000 e o vagão QT105566.

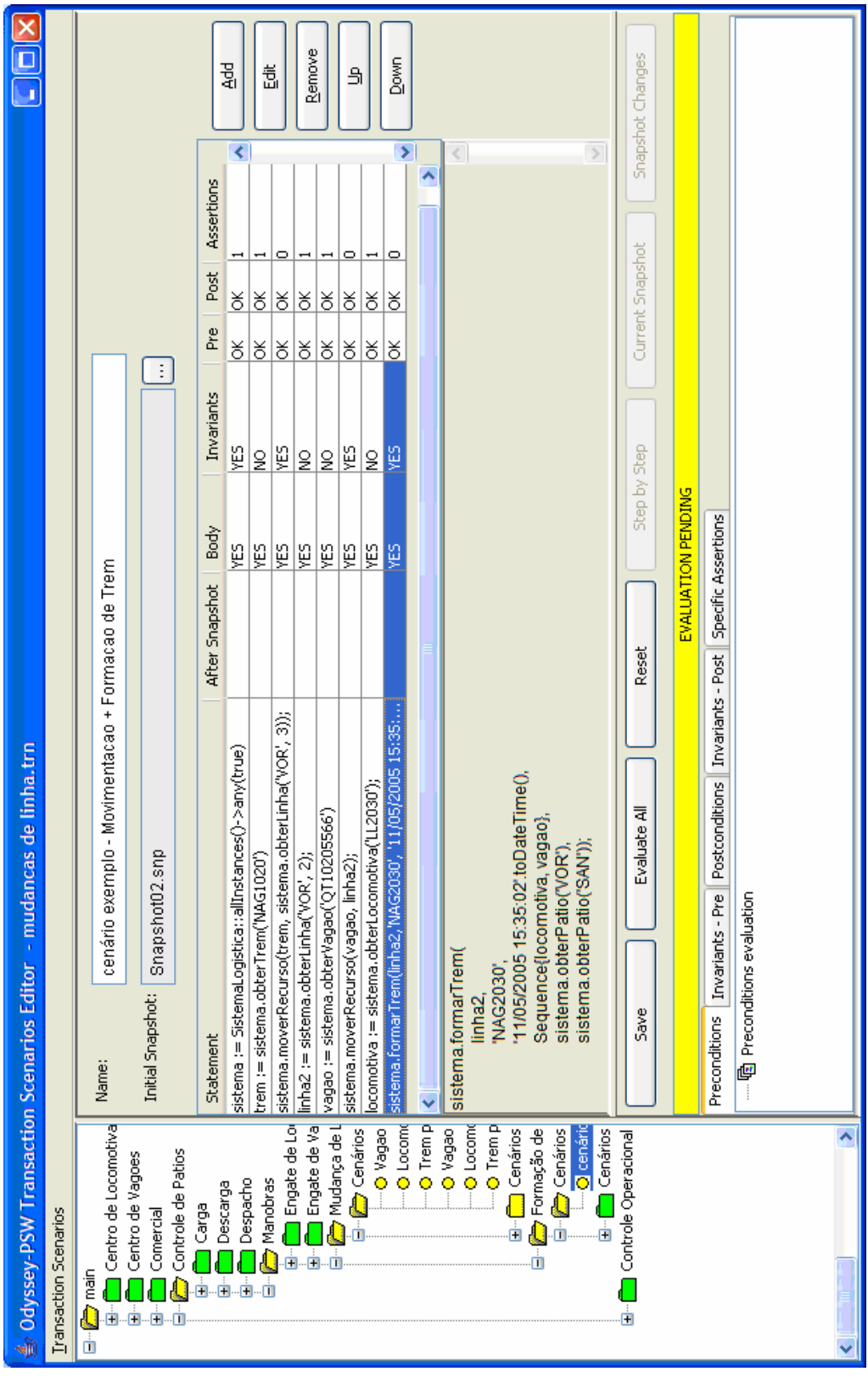


Figura 8.33 - Exemplo de avaliação de uma seqüência de operações

Essas três operações correspondem, respectivamente, às seguintes invocações de operações:

a) sistema.moverRecurso(trem, sistema.obterLinha('VOR', 3))

b) sistema.moverRecurso(vagao, linha20)

c) sistema.formarTrem (linha2, 'NAG2030', '11/05/2005 15:35:00'. toDateTime())

Essas invocações de operações utilizam variáveis definidas em passos intermediários do cenário como, por exemplo, *trem*, *linha2*, *vagao* e *locomotiva*. Dessa forma, é possível, em uma chamada de operação, utilizar o resultado de operações executadas anteriormente na seqüência. A Figura 8.34 ilustra como a chamada *vagao := sistema.obterVagao('QT10205566')* presente na quinta linha da sequência apresentada na Figura 8.33 é definida.

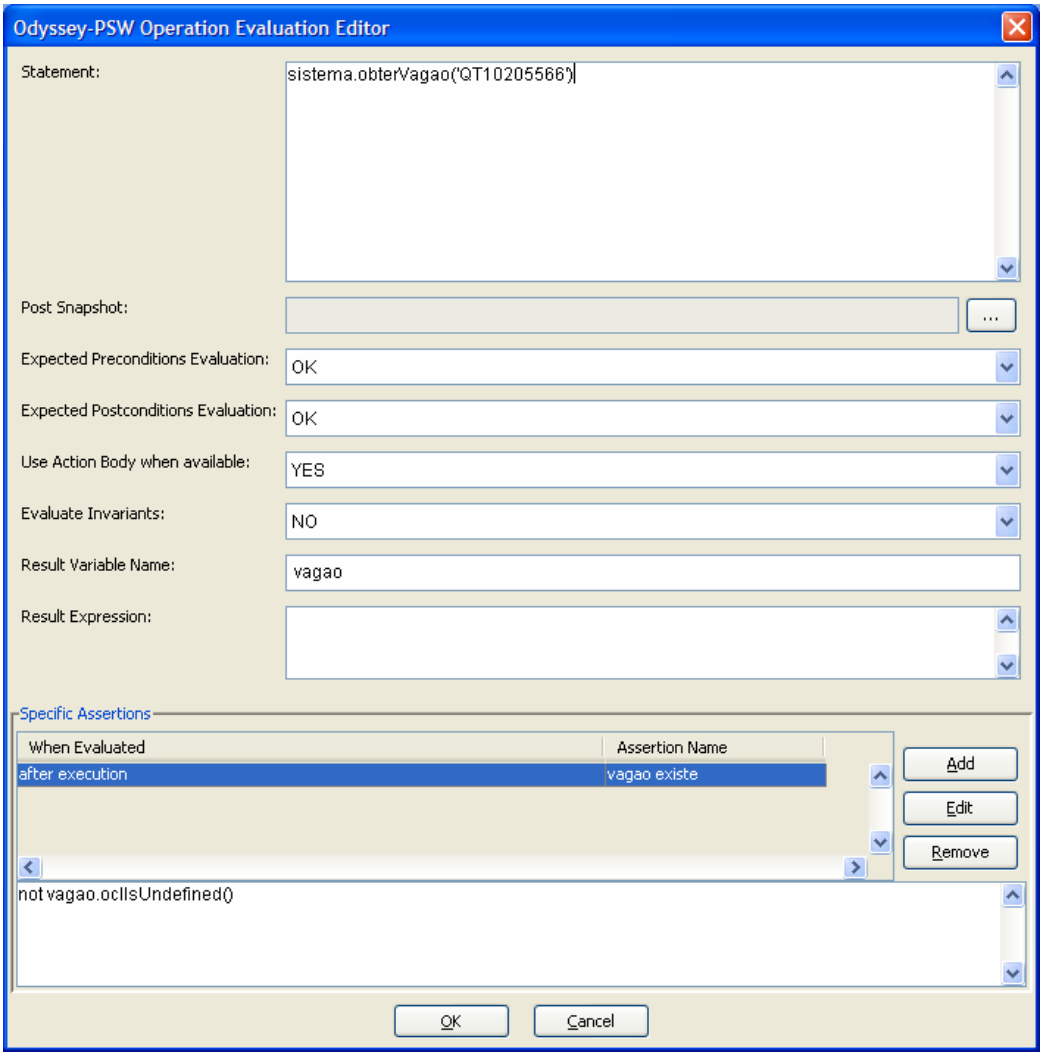


Figura 8.34 - Exemplo de definição de variáveis reutilizáveis

O resultado de *sistema.obterVagao('QT10205566')* será armazenado na variável *vagao* (campo *Result Variable Name*), podendo ser utilizado nas chamadas subseqüentes da seqüência, como ocorre, por exemplo, na última chamada (*sistema.formarTrem*). Na parte inferior da Figura 8.34, assertivas específicas associadas com uma invocação de operação do cenário podem ser especificadas através de expressões OCL. Nesse exemplo, a assertiva verifica, após a execução da operação, se existe de fato um vagão com aquele número de série.

É possível executar a avaliação de um cenário específico através do botão *EvaluateAll* no painel de edição do mesmo (Figura 8.33), ou de todos os casos de teste de um grupo, selecionando o grupo no painel esquerdo e ativando a opção *Evaluate All* no menu. Se o cenário de avaliação for composto por uma seqüência de chamadas a operações, o botão *Step by Step* pode ser utilizado para fazer a execução de uma chamada por vez. Além disso, o botão *Current Snapshot* ativa o módulo gerenciador de espaço de objetos com o espaço de objetos corrente, permitindo a visualização das instâncias resultantes da execução de um passo específico ou de toda a seqüência de chamadas. O botão *Reset* permite reiniciar o procedimento de avaliação da seqüência de operações.

A Figura 8.35 apresenta a janela resultante da avaliação de casos de teste. Todos os casos de teste cujas avaliações resultem nos valores esperados são indicados na árvore pela cor verde, enquanto que os demais casos de teste avaliados são indicados pela cor vermelha. O resultado resumido da avaliação de cada chamada do cenário pode ser visualizado através das colunas *Pre*, *Post* e *Assertions*, que indicam, através desse mesmo esquema de cores, se os resultados esperados para as pré-condições, pós-condições e assertivas específicas foram obtidos. A coluna *Pre* corresponde à avaliação de todas as precondições da operação invocada, que pode ser combinada com a avaliação das restrições globais do modelo (invariantes), dependendo da configuração do campo *Evaluate Invariants* descrito anteriormente. De forma análoga, a coluna *Post* corresponde à avaliação de todas as pós-condições da operação invocada. Finalmente, a coluna *Assertions* corresponde à avaliação de todas as assertivas específicas associadas com esse passo do cenário.

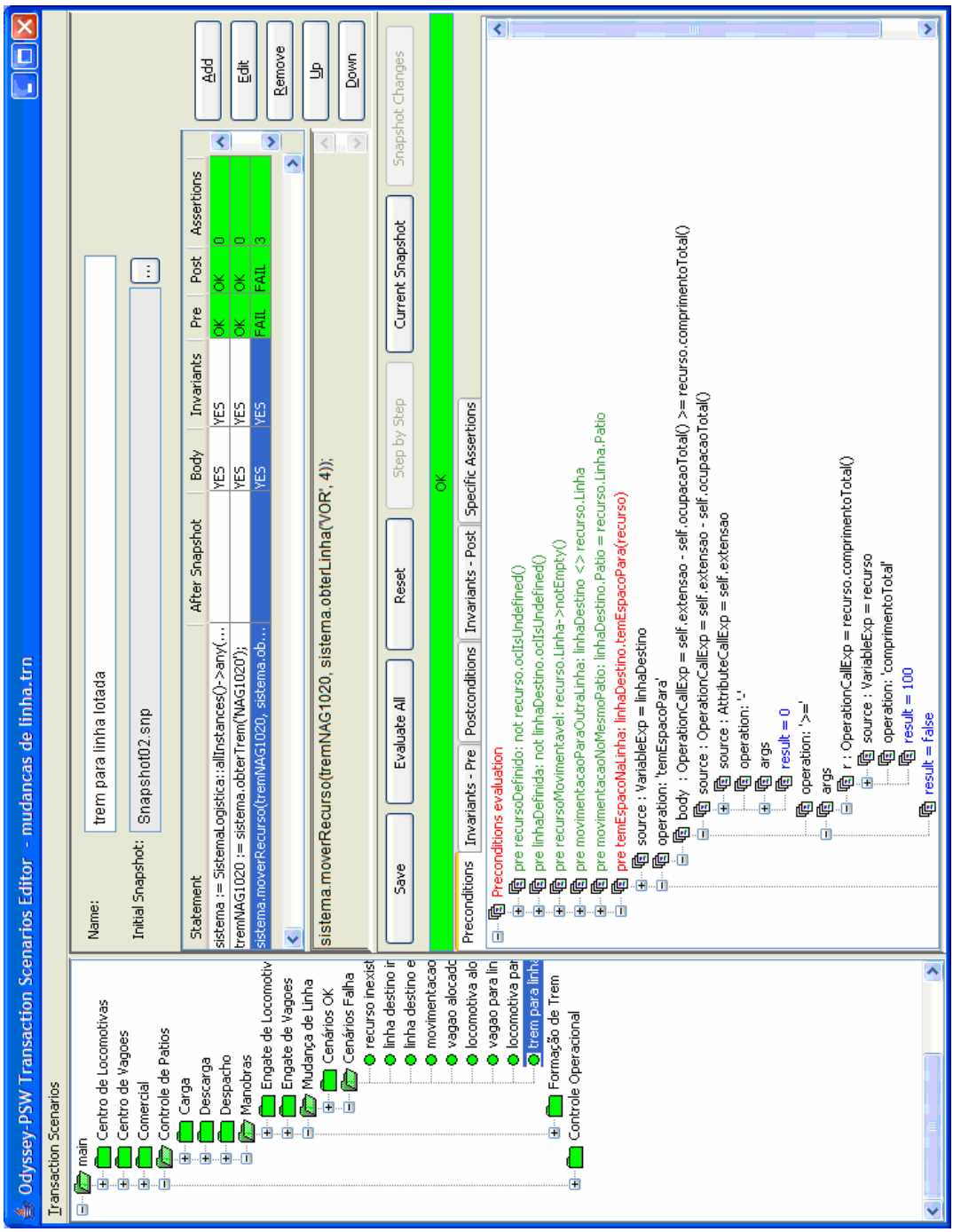


Figura 8.35 – Janela de resultado da avaliação de um cenário

Além da visão resumida, é possível também visualizar o resultado de forma detalhada no painel inferior, que é dividido em cinco abas:

- *Preconditions*: apresenta, no formato *AST View* descrito anteriormente, a avaliação de todas as precondições associadas à operação selecionada. O exemplo da Figura 8.35 corresponde a um cenário onde é esperada uma falha

na avaliação das precondições. O painel inferior indica, através da cor vermelha, que a precondição *temEspacoNaLinha* não foi atendida, pois o espaço disponível na linha destino era igual a 0 no momento da invocação.

- *Invariants-Pre*: apresenta as restrições do modelo que foram violadas considerando-se o espaço de instâncias imediatamente anterior à execução da operação.
- *Postconditions*: apresenta a avaliação de todas as pós-condições associadas à operação, de forma análoga às precondições.
- *Invariants-Pos:* apresenta as restrições do modelo que foram eventualmente violadas considerando-se o estado de objetos após a execução da operação.
- *Assertions*: apresenta as assertivas específicas associadas à chamada da operação cujas avaliações não resultem no valor verdadeiro.

As assertivas específicas podem ser utilizadas, por exemplo, para refinar o processo de avaliação das pré-condições, evitando a obtenção de uma falsa indicação de que o resultado obtido foi o esperado. A Figura 8.36 apresenta um exemplo onde são definidas três assertivas específicas que serão utilizadas no cenário apresentado na Figura 8.35. Cada assertiva é uma expressão OCL que será avaliada imediatamente antes ou após a execução da operação, conforme especificado pelo campo "*When Evaluated*". As assertivas indicadas para serem executadas antes da execução da operação (coluna *When Evaluated* com o valor *before execution)* visam verificar que, no momento da chamada da operação especificada no campo *"Statement",* o espaço de instâncias é tal que todas as pré-condições da operação *obterLinha*, exceto *temEspacoNaLinha*, são verdadeiras.

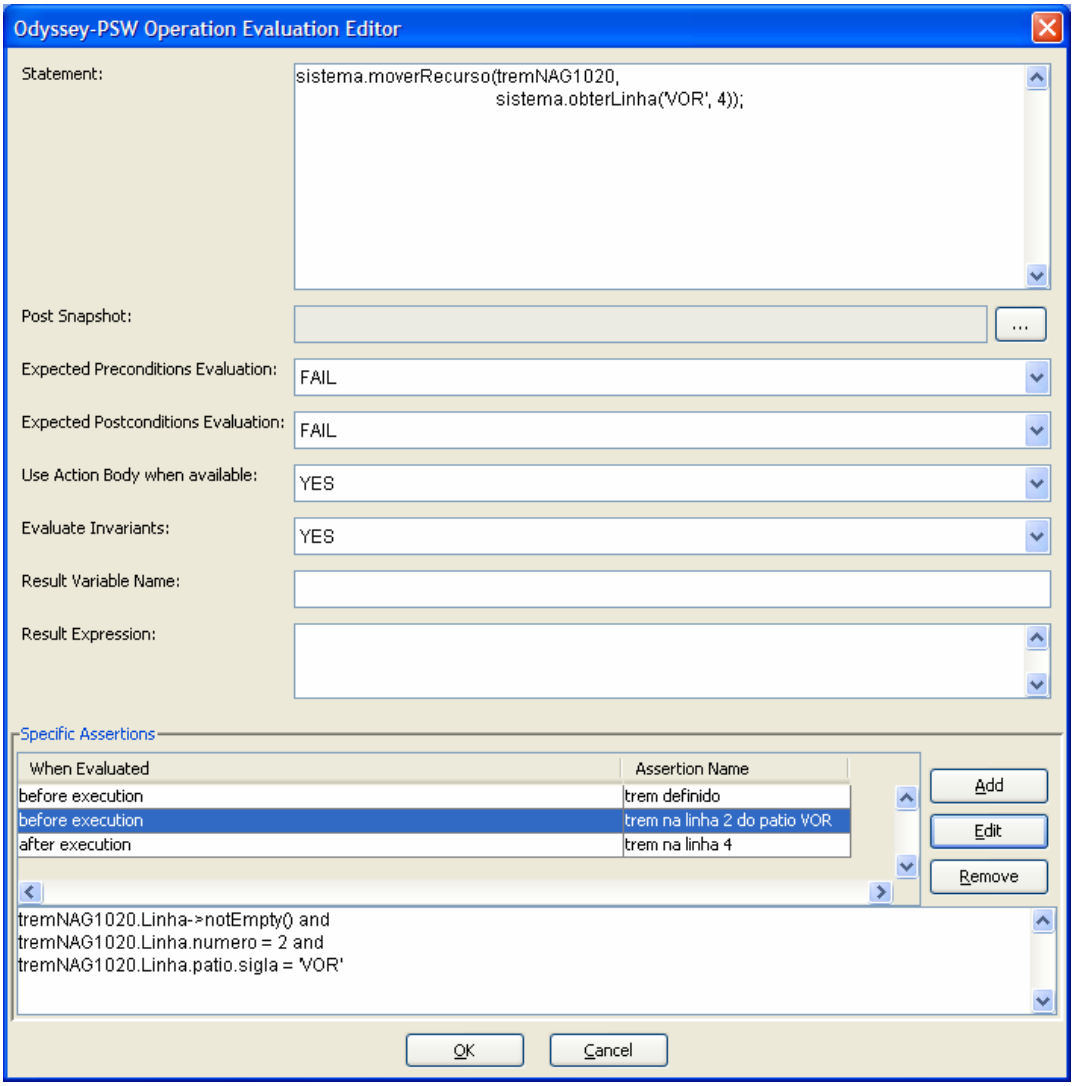


Figura 8.36- Exemplo de assertivas específicas

Uma característica importante do Odyssey-PSW é a capacidade de avaliar as restrições ligadas ao escopo da operação, definidos através da cláusula *modifiable*, descrita no capítulo 5. A Figura 8.37 apresenta um exemplo no qual a operação *moverRecurso*, descrita na Figura 8.30, passou a ter o escopo definido por duas cláusulas *modifiable*. Estas cláusulas definem que somente as ligações entre a *linhaDestino* e os *recursosEstacionados* nessa linha, e entre a linha anteriormente associada ao recurso movido e os recursos que estavam estacionados nela, podem ser modificadas.

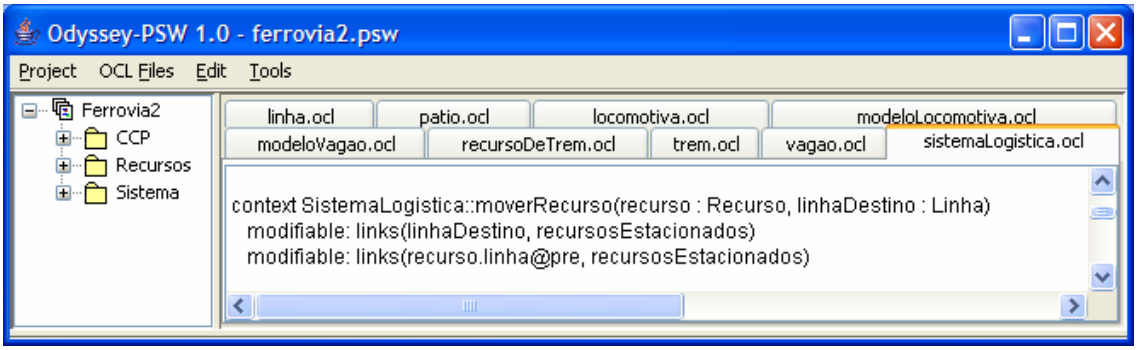


Figura 8.37 – Exemplo de especificação de uma cláusula *modifiable*

A Figura 8.38 apresenta as ações associadas à operação *moverRecurso*. As duas primeiras ações geram efeitos que não estão previstos no escopo da operação. Apenas a última ação, que modifica a ligação entre o recurso e a sua linha, é permitida pelas cláusulas *modifiable* associadas à operação.

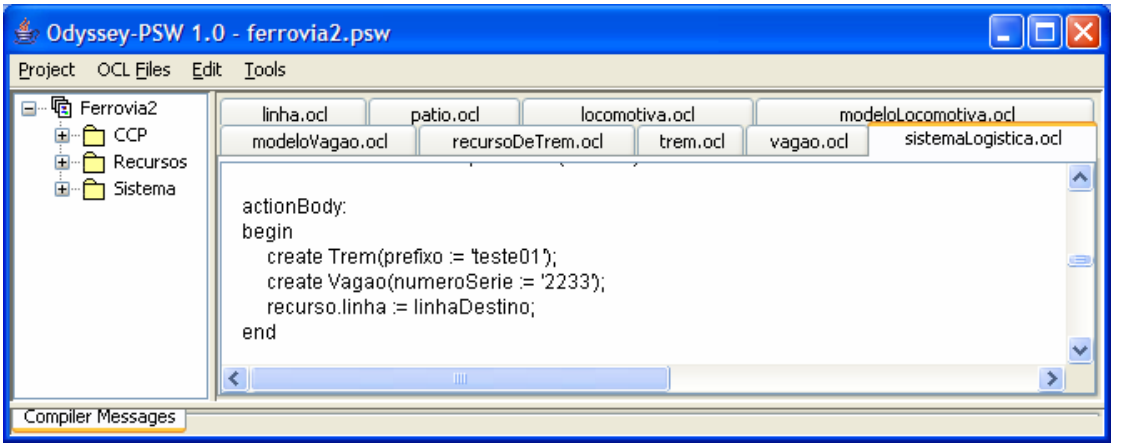


Figura 8.38 - Exemplo de ações que violam o escopo permitido para a operação

Se um cenário de movimentação de um recurso de uma linha para outra for avaliado, considerando o conjunto de ações especificado para a operação *moverRecurso,* o resultado indicará falha nas pós-condições dessa operação, pois ocorrerão modificações fora do escopo permitido por essa operação. Essa falha será indicada pela mensagem "*changes not allowed by frame constraints have been detected"*, que pode ser visualizada na parte inferior da Figura 8.39.

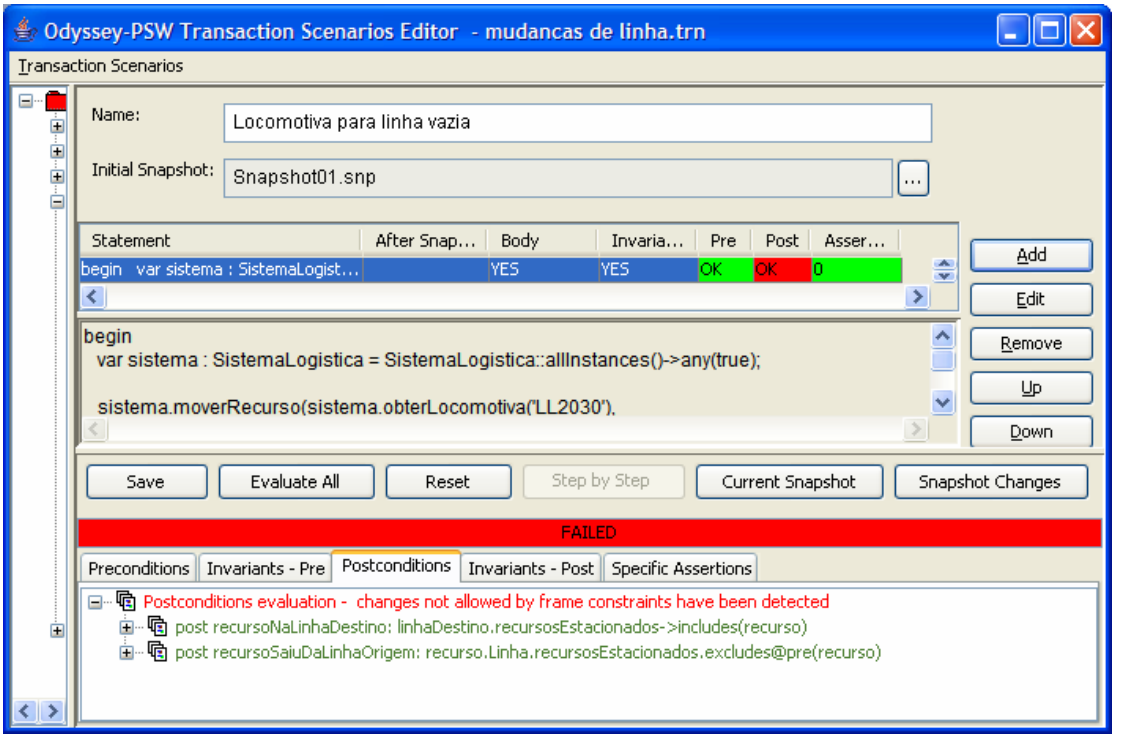


Figura 8.39 – Exemplo de violação da pós-condição em função de modificação fora do escopo permitido

Pressionando o botão *SnapshotChanges*, o usuário pode visualizar todas as modificações que ocorreram após a execução do cenário completo ou de um passo específico. As modificações que tiverem violado o escopo da operação, definido pelas suas cláusulas *modifiable,* são apresentadas em vermelho. A Figura 8.40 apresenta um exemplo para o cenário ilustrado pela Figura 8.39. Essa janela informa que duas instâncias foram criadas (um trem e um vagão), e como estão em vermelho, correspondem a modificações não permitidas pelo escopo da operação *moverRecurso*. As outras modificações informadas correspondem à criação de uma ligação entre a locomotiva e a linha destino, e a eliminação da ligação entre a locomotiva e a linha de onde ela foi movimentada.

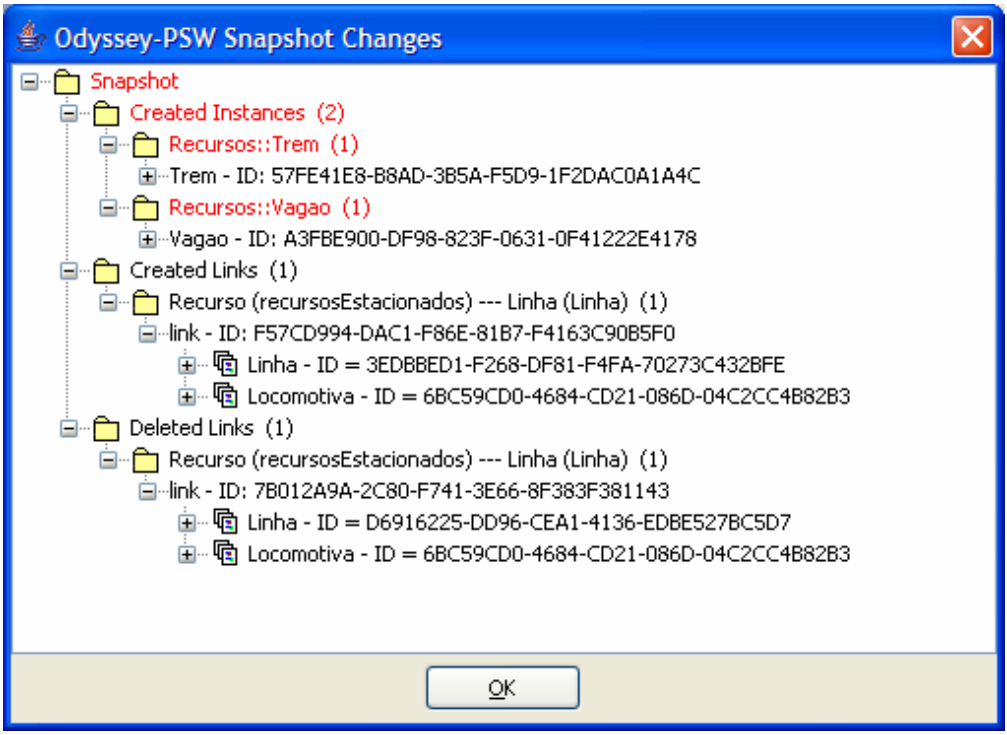


Figura 8.40 – Exemplo de modificações detectadas após a execução de um cenário ou passo

8.7 Aplicação de Reestruturações ao Modelo

A criação de instâncias e a formalização de algumas regras do modelo utilizado como exemplo ao longo deste capítulo indicam que algumas reestruturações podem ser aplicadas a esse modelo. Por exemplo, a redundância de informações presente nas instâncias da classe *Vagão* ilustradas pela Figura 8.5 e a consulta apresentada na Figura 8.18 indicam a existência de um conceito referente a tipos de vagão (cimento, minério, etc) que não foi representado pelas classes da primeira versão do modelo (Figura 8.1). De forma análoga, isto ocorre com as locomotivas e o conceito de tipos de locomotiva. As restrições associadas à classe *Trem,* apresentadas na Figura 8.3, poderiam ser simplificadas se fosse criada uma generalização para as classes *Vagão* e *Locomotiva*. Além disso, a relação da classe *Trem* com seus recursos (*Vagão* e *Locomotiva*) não captura o fato de que os recursos alocados a um trem apresentam uma ordem seqüencial.

A Figura 8.41 apresenta uma versão reestruturada do modelo obtida através da aplicação das seguintes operações básicas de reestruturação:

- criar a classe *ModeloVagao*;
- criar uma associação entre as classes *Vagao* e *ModeloVagao*;
- mover os atributos *capacidade*, *comprimentoTesteiras* e *comprimentoEngates* para a classe *ModeloVagao*;
- adicionar a operação *comprimentoTotal* à classe *ModeloVagao;*
- criar a classe *ModeloLocomotiva*;
- mover os atributos *tracao* e *capacidade* para *ModeloLocomotiva*;
- criar uma associação entre *Locomotiva* e *ModeloLocomotiva*;
- adicionar a operação *comprimentoTotal* à classe *ModeloLocomotiva*;
- criar uma classe *RecursoDeTrem*;
- definir a classe *RecursoDeTrem* como especialização da classe *Recurso*;
- definir as classes *Locomotiva* e *Vagão* como especializações da classe *RecursoDeTrem*;
- definir uma associação entre *Trem* e *RecursoDeTrem*;
- remover a associação entre a classes *Trem* e *Vagão*;
- remover a associação entre a classes *Trem* e *Locomotiva*;

Além dessas reestruturações, dois novos elementos foram adicionados ao modelo:

- uma generalização das classes *ModeloVagao* e *ModeloLocomotiva,* denominada *ModeloRecursoTrem,* contendo um atributo *nome.*
- a restrição {ordered} na associação entre *Trem* e *RecursoTrem*, para indicar que um trem é formado por um conjunto ordenado de recursos.

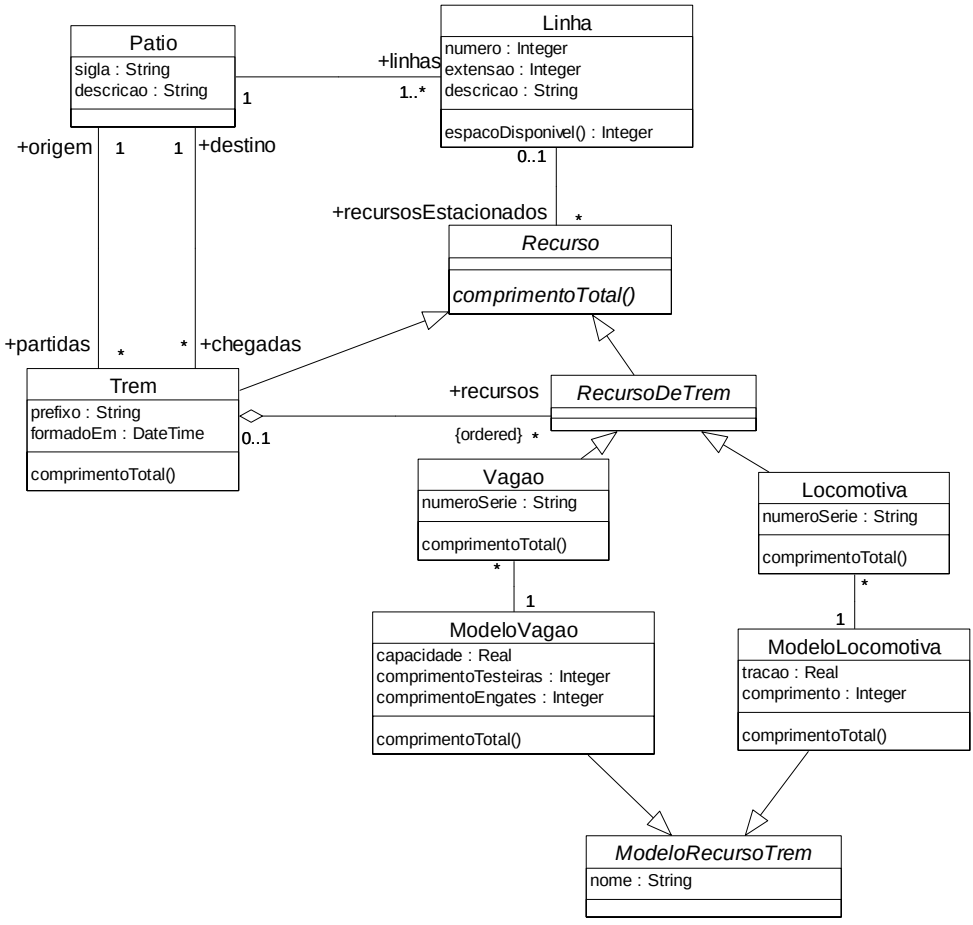


Figura 8.41 – Modelo da ferrovia após a reestruturação

Essas reestruturações no modelo resultam em modificações nas restrições especificadas em OCL, tais como as descritas abaixo:

- mover a definição do corpo da operação *comprimentoTotal* da classe *Vagao* para a classe *ModeloVagao*.

  context ModeloVagao::comprimentoTotal() : Integer

  body: comprimentoTesteiras + comprimentoEngates

  context Vagao::comprimentoTotal() : Integer

  body modeloVagao.comprimentoTotal()

- mover a definição do corpo da operação *comprimentoTotal* da classe *Locomotiva* para a classe *ModeloLocomotiva*.

  context ModeloLocomotiva::comprimentoTotal() : Integer

  body: comprimento

  context Locomotiva::comprimentoTotal() : Integer

  body: modeloLocomotiva.comprimentoTotal()

- mover a restrição duplicada nas classes *Trem* e *Locomotiva* para a superclasse *RecursoDeTrem*

  context RecursoDeTrem

  inv: trem->notEmpty() xor linha->notEmpty()

- substituir o corpo da operação *comprimentoTotal* da classe *Trem* de modo a utilizar a nova associação com a classe *RecursosTrem*. As definições originais dessa classe estão na Figura 8.3.

  context Trem::comprimentoTotal() : Integer

  -- comprimento total de um trem é dado pela soma do comprimento total

  -- dos recursos que compõem um trem, i.e., seus vagões e locomotivas.

  body: recursos.comprimentoTotal()->sum()

- definir uma propriedade denominada *vagões* que corresponde ao subconjunto dos recursos de trem do tipo *Vagao*.

  context Trem

  def: vagoes : Set(Vagao) =

  recursos->select(oclIsKindOf(Vagao)).oclAsType(Vagao)->asSet()

- definir uma propriedade denominada *locomotivas* que corresponde ao subconjunto dos recursos de trem do tipo *Locomotiva*.

  def: locomotivas : Set(Locomotiva) =

  recursos->select(oclIsKindOf(Locomotiva)).oclAsType(Locomotiva)->asSet()

- Remover da classe *Trem* as definições referentes às operações *comprimentoTotalVagoes* e *comprimentoTotalLocomotivas*.

Uma vez que estas modificações tenham sido realizadas, os espaços de objetos e respectivos scripts de criação, bem como os casos de teste de avaliação de operações de consulta ou modificadoras devem ser adaptados à nova estrutura do modelo. A Figura 8.42 apresenta um exemplo que ilustra como as instâncias de vagão passaram a ser representadas após a adaptação ao novo modelo.

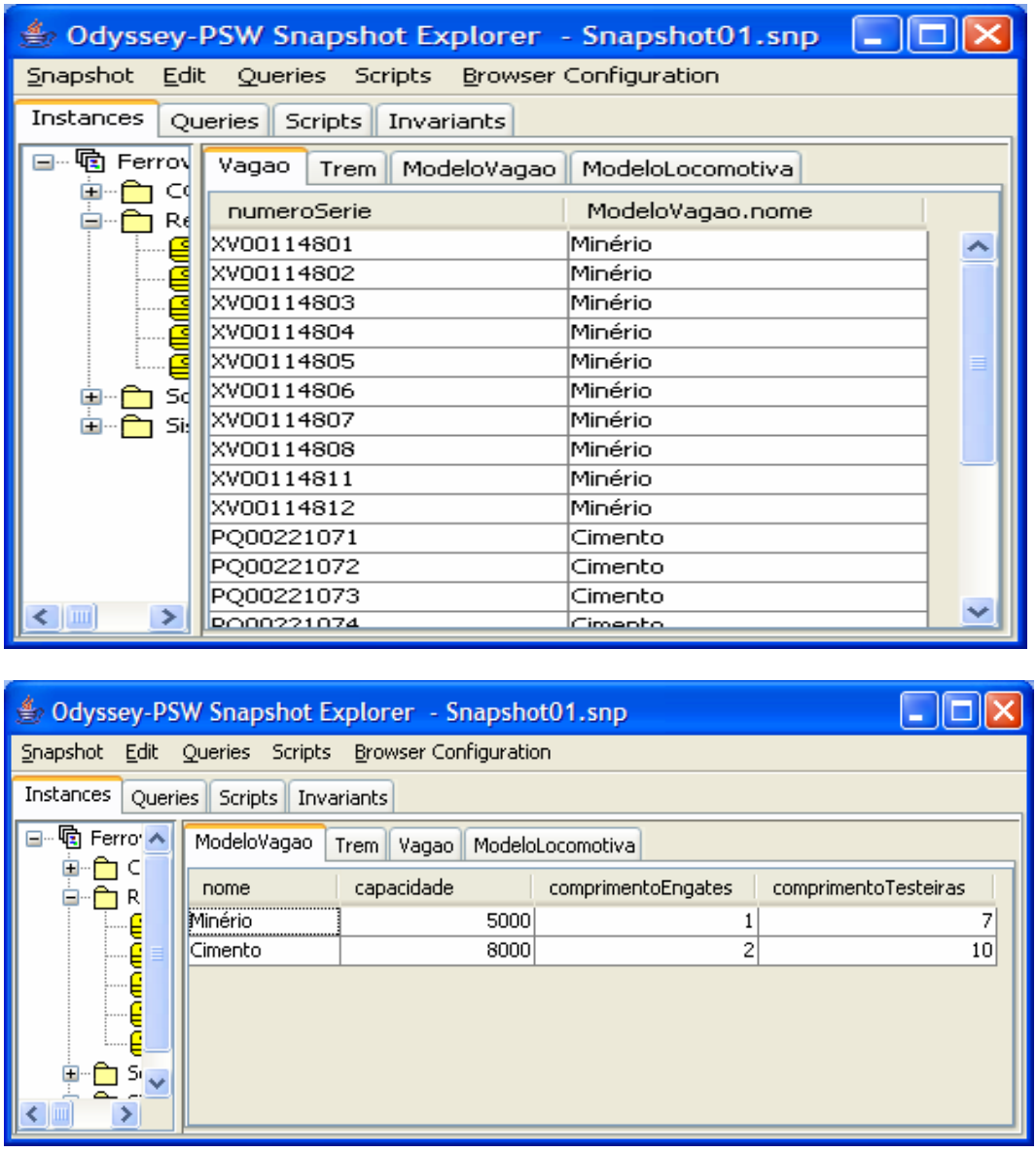


Figura 8.42 - Instâncias de Vagão e ModeloVagao após a reestruturação do modelo

8.8 Considerações Finais

Este capítulo descreveu o software que apoia a realização de algumas atividades de verificação e validação de um modelo com restrições especificadas em OCL, bem como pode apoiar a avaliação da preservação da semântica do modelo após a realização de reestruturações.

Existem diversas ferramentas que oferecem algum apoio à produção de modelos com restrições OCL (TOVAL et al., 2003). A maior parte oferece apoio apenas para a avaliação sintática e a verificação de tipos das expressões OCL. Poucas oferecem recursos para apoiar a validação do modelo. ModelRun (BORLAND INC., 2002) é uma ferramenta que oferece uma forma interativa de criar e manipular objetos de um modelo UML, e avalia se alguma restrição (invariante) definida no modelo é violada em uma determinada configuração de objetos. Suas funcionalidades são similares àquelas descritas no módulo gerenciador de espaços de objetos do Odyssey-PSW. Entretanto, a ferramenta ModelRun é baseada na OCL 1.0 e, portanto, não permite a utilização de diversos recursos da OCL 2.0, como tuplas e construções *def*, por exemplo. Além disso, ela é limitada à definição e avaliação de invariantes.

OCLE (CHIOREAN et al., 2004) é uma ferramenta que permite a criação de instâncias de um modelo UML e a avaliação de invariantes sobre essas instâncias. Um outro recurso oferecido por essa ferramenta é a geração de código Java para as especificações de operações de consulta efetuadas através da construção *body*. A OCLE é compatível com a OCL 2.0, e embora analise sintaticamente as pré e pós-condições das operações, não oferece nenhum recurso para a sua validação. De forma similar à estrutura de projeto do Odyssey-PSW, a OCLE pode avaliar expressões OCL tanto em um espaço de objetos no nível M0 como no nível M1.

USE (RICHTERS e GOGOLLA, 2000) é a ferramenta mais diretamente relacionada ao Odyssey-PSW. De forma similar às outras ferramentas descritas nesta seção, ela é capaz de avaliar violações de invariantes sobre um espaço de objetos. Esse espaço de objetos é criado a partir de uma linguagem de script que oferece alguns comandos como, por exemplo, para criar ou destruir objetos, criar ou destruir ligações entre objetos, e para modificar o valor de um atributo de um objeto. Entretanto, essa linguagem não possui estruturas de controle ou de repetição. A ferramenta oferece

recursos para animar uma especificação, que combina esses comandos da linguagem de script com comandos de sinalização de entrada e saída de uma operação. As pré-condições da operação são avaliadas após uma sinalização de entrada na operação, enquanto que as pós-condições são avaliadas após uma sinalização de saída. Entretanto, a avaliação dos resultados esperados em cada animação deve ser realizada visualmente através das mensagens exibidas na tela. Ao contrário do Odyssey-PSW, ela não oferece recursos para a definição dos resultados esperados, nem para a execução automática de testes de regressão a partir de um conjunto de casos de teste. Além disso, o USE possui outras limitações, como por exemplo: não oferece recurso para importação de modelos produzidos em outras ferramentas, ou seja, o modelo deve ser elaborado em uma linguagem textual específica; a avaliação das pré e pós-condições de uma operação que redefine uma operação de uma superclasse não é capaz de detectar violações como pré-condições mais fortes ou pós-condições mais fracas. Por outro lado, o USE possui alguns recursos interessantes que não são oferecidos pelo Odyssey-PSW, como por exemplo, a geração de um diagrama de seqüência com a representação gráfica das interações ocorridas em um cenário de animação, e a visualização dos objetos na forma de um diagrama de objetos.

Uma alternativa considerada para a implementação foi a transformação da OCL para outras linguagens como Alloy ou VDM++, de forma a reutilizar as ferramentas disponíveis para essas linguagens, que oferecem apoio para atividades de verificação e validação de especificações. Entretanto, como o Alloy tem um poder de expressão menor do que a OCL, e como a interpretação dos resultados da avaliação das expressões OCL teria que ser feito em uma outra linguagem (Alloy ou VDM++), julgamos mais conveniente produzir uma implementação específica para a OCL e para as extensões propostas nesta tese.

A implementação de reestruturações automáticas foi realizada apenas para algumas reestruturações simples (Cadeia de Implicações, Cadeia de Quantificadores Universais) como prova de conceito da arquitetura da ferramenta. Essa implementação opera apenas com a versão já compilada das expressões. Para integrá-las à versão final da ferramenta, seria necessário desenvolver um conversor de expressões da versão compilada para a sua sintaxe concreta, e esta implementação não fez parte do escopo desta tese. Desta forma, todas as reestruturações do exemplo descrito na seção 8.7 foram

realizadas manualmente. Após cada reestruturação, os casos de teste foram executados como forma de avaliação da preservação da semântica do modelo.

Capítulo 9

Conclusões

Este capítulo resume o trabalho efetuado (seção 9.1) e lista suas principais contribuições para a área de Engenharia de Software (seção 9.2). Além disso, discutimos as principais limitações do atual estado de desenvolvimento do trabalho (seção 9.3), e apontamos algumas perspectivas de trabalhos futuros (seção 9.4).

9.1 Resumo do Trabalho

Conforme ressaltamos no capítulo 2, embora a UML ofereça um grande número de diagramas que possibilitam a construção de visões estáticas e dinâmicas de um sistema, eles não são suficientes para descrever todos os detalhes que compõem um modelo de software ou mesmo um metamodelo. Restrições, regras, definições contratuais de operações são alguns exemplos de informações que não são cobertas por esses diagramas (WARMER e KLEPPE, 2003). Em cenários que demandam maior precisão do modelo, a OCL é uma linguagem que pode ser utilizada para especificar, de forma precisa, elementos que as notações gráficas da UML não são capazes de representar, como, por exemplo, restrições, expressões associadas a atributos derivados e expressões de consulta.

Embora a OCL tenha sido definida com o objetivo de ser uma linguagem de uso mais fácil, se comparada às linguagens formais tradicionais (WARMER e KLEPPE, 2003), especificações produzidas com a OCL podem apresentar problemas de legibilidade e manutenibilidade, em função da presença de construções inadequadas, seja nas expressões OCL, seja no modelo subjacente.

A partir de um estudo realizado em diversos trabalhos, como por exemplo, em especificações produzidas por alunos de um curso de pós-graduação em desenvolvimento de software do NCE-UFRJ, nas especificações da UML 1.x, UML 2.0 (superestrutura e infra-estrutura), MOF 1.4 e OCL 2.0, bem como em artigos científicos internacionais contendo restrições expressas em OCL, identificamos e catalogamos diversas construções potencialmente problemáticas ao entendimento e à manutenção de

uma especificação. Essas construções, denominadas *OCL Smells,* assim como um conjunto de reestruturações propostas para removê-las, foram descritas nos capítulos 3 e 4. A execução de um estudo experimental, descrito no capítulo 5, nos permitiu fazer uma avaliação inicial de como a compreensão de restrições especificadas em OCL pode ser afetada pela estrutura das expressões que a compõem. Os resultados desse estudo indicam que, na população analisada, a estrutura das expressões OCL pode exercer influência no desempenho dos participantes em relação à correção do entendimento e ao tempo despendido para chegar a um entendimento sobre uma restrição OCL. Expressões contendo *OCL smells* tiveram um menor índice de acerto em relação à correção do entendimento e os participantes gastaram mais tempo para entendê-las.

O capítulo 6 apresentou a abordagem proposta para apoiar a automação de reestruturações aplicadas a modelos com restrições expressas em OCL. Essa abordagem é baseada na definição de operações de transformação, onde as condições para a aplicação de uma reestruturação, bem como os efeitos esperados, são especificadas em OCL, enquanto que as ações de transformação são definidas em uma linguagem de ações definida nesta tese, a OCL-AL (OCL Action Language), que estende a OCL de modo a permitir a elaboração de especificações executáveis de operações.

Uma questão relevante no contexto de transformações aplicadas a modelos é garantir a preservação da semântica do modelo após uma reestruturação, especialmente quando ela é realizada de forma manual. A estratégia proposta nesta tese consiste em empregar testes de regressão executados de forma automatizada, reutilizando os casos de teste definidos para apoiar a avaliação da semântica do modelo. Vale ressaltar, entretanto, que essa estratégia deve ser combinada com outras formas de verificação, como por exemplo, análises formais e revisões, visto que os testes não provam que um modelo está semanticamente correto, nem que a versão reestruturada de um modelo é semanticamente equivalente a sua versão anterior.

O capítulo 7 descreveu um conjunto de extensões à OCL que julgamos necessárias para a implementação e utilização prática da abordagem proposta. Finalmente, o capítulo 8 descreveu o software Odyssey-PSW, cujo propósito é apoiar a verificação e validação da semântica de um modelo com restrições especificadas em OCL, assim como a avaliação da preservação da semântica do modelo após a realização de reestruturações, a partir da execução de testes de regressão.

9.2 Contribuições

Dentre as contribuições desta tese, podemos destacar:

- A identificação de diversos tipos de construção encontrados em modelos com restrições especificadas em OCL que podem contribuir negativamente para o seu entendimento e a sua evolução. Os *OCL Smells* mais freqüentemente encontrados foram registrados de forma a poderem ser utilizados não apenas na identificação de possíveis alvos de reestruturação em modelos já existentes, mas também para evitar que essas construções sejam inseridas em novos modelos.

- A definição de um conjunto de reestruturações que podem ser aplicadas tanto nas expressões OCL, como no modelo associado, de forma a substituir construções potencialmente problemáticas de especificações elaboradas em OCL por outras mais adequadas. Julgamos que tanto os *OCL Smells* como as reestruturações propostas nesta tese registram um conhecimento importante para a comunidade envolvida na produção de especificações de modelos, de regras de boa formação de meta-modelos, ou de qualquer outra atividade que envolva a produção de expressões ou restrições em OCL.

- O planejamento e a execução de um estudo experimental para análise da viabilidade de aplicação das reestruturações e do seu impacto no entendimento de especificações produzidas com a OCL. Todos os instrumentos utilizados para o seu planejamento e execução estão disponíveis para futuras repetições.

- A definição de uma abordagem de apoio para a automação de reestruturações aplicadas a modelos e restrições em OCL.

- A definição de uma abordagem de integração da OCL com a semântica de ações definida na especificação da UML, de modo a possibilitar a elaboração de modelos executáveis de sistemas, bem como a automação de pequenas transformações em modelos, tais como reestruturações. Essa integração resultou na definição da linguagem OCL Action Language.

- A definição de algumas extensões à OCL que visam lidar com algumas limitações e deficiências existentes na linguagem. Em particular, foram definidas construções que permitem a definição do escopo das alterações

admissíveis por uma operação, e que possibilitam a reutilização de expressões em diferentes restrições associadas a uma operação.

- A construção da ferramenta Odyssey-PSW que oferece apoio à realização de atividades de verificação sintática e de tipos, de validação de modelos UML e restrições especificadas em OCL, bem como às reestruturações realizadas manualmente, através da execução automática de testes de regressão.

Nos últimos anos, as técnicas de reestruturação passaram a ser aplicadas em artefatos mais abstratos, tais como modelos, por exemplo. O foco de sua aplicação, porém, se restringia aos diagramas UML, como descrito em (SUNYÉ et al., 2001), onde foram propostas algumas reestruturações aplicáveis a diagramas de classes e diagramas de estados de um modelo UML, em (BOGER et al., 2002), onde foi descrita a aplicação de algumas reestruturações a diagramas de classes, diagramas de estados e de atividades, e em (PORRES, 2003), onde foram propostas algumas reestruturações que podem ser aplicadas a diagramas UML.

Entretanto, nenhum trabalho do nosso conhecimento tinha abordado problemas relacionados à estruturação inadequada de especificações produzidas em OCL, e às possíveis reestruturações que podem ser aplicadas neste contexto. A relevância desta tese é reforçada pelo fato desses problemas serem encontrados com freqüência em especificações que utilizam a OCL como, por exemplo, as várias versões da UML e alguns trabalhos publicados pela comunidade científica. Pelos resultados obtidos no nosso estudo, acreditamos que tenhamos dado um primeiro passo em direção à produção de especificações em OCL de melhor qualidade.

A utilização da OCL e da OCL-AL para a especificação e a execução de reestruturações nos permitiu identificar diversos pontos em que a OCL pode evoluir, além de possíveis pontos de convergência entre a OCL e a UML Action Semantics. Acreditamos que esta tese tenha trazido algumas contribuições não apenas para a evolução e o amadurecimento da OCL, mas também para a sua utilização prática, em função do apoio oferecido pelo software que implementamos, descrito no capítulo 7. O apoio automatizado é uma questão importante na utilização prática de linguagens como a OCL, e a oferta de soluções nesta área é ainda bastante limitada (BAAR et al., 2005).

9.3 Limitações

Esta seção discute algumas limitações que foram consideradas no desenvolvimento desta tese:

O conjunto de *OCL smells* proposto nesta tese não representa uma lista exaustiva de todas as construções potencialmente problemáticas que podem ocorrer em uma especificação, mas sim daquelas que foram encontradas com maior freqüência nas especificações que foram objeto deste estudo. O mesmo ocorre com o conjunto de reestruturações. Na verdade, os dois catálogos são documentos que devem estar em constante evolução. Uma outra limitação deste trabalho reside no fato das reestruturações e das provas de preservação da semântica para cada reestruturação não terem sido formalmente definidas.

Algumas limitações estão relacionadas ao estudo experimental apresentado no capítulo 5. As inferências estatísticas foram realizadas em uma população pequena, composta de 22 participantes. É reconhecido que quanto maior o número de participantes em um estudo experimental, maior será sua capacidade de conclusão. Assim, para que as conclusões do estudo possam ser estabelecidas para uma população maior, com maior grau de certeza, o estudo deve ser repetido com um maior número de participantes e em diferentes contextos.

Outra limitação do estudo experimental refere-se a sua concentração apenas nos aspectos de entendimento de restrições do tipo invariante, onde procuramos identificar os efeitos gerais que a estrutura das restrições poderiam gerar no entendimento. Outros estudos experimentais devem ser realizados para analisar outras facetas das propostas apresentadas nesta tese, como por exemplo, estudos específicos sobre cada tipo de *OCL Smell* e de reestruturação, bem como sobre as abordagens propostas para apoio automatizado.

A abordagem utilizada para apoiar a verificação da preservação da semântica do modelo após as reestruturações foi baseada na utilização de testes de regressão, reutilizando-se os casos de teste definidos para apoiar a validação da semântica das restrições. Uma vez que os testes não provam que um modelo está semanticamente correto, nem que a versão reestruturada de um modelo é semanticamente equivalente à sua versão anterior, eles devem ser combinados com outras estratégias de verificação. Entretanto, nossa estratégia inicial de automação foi aceitar um nível de garantia menor,

em troca de um resultado mais rápido e de uma menor dependência em relação a conhecimentos de provas formais por parte dos usuários.

Como qualquer estratégia baseada em análise dinâmica, o nível de confiança obtido está diretamente relacionado com a qualidade dos casos de teste utilizados. Entretanto, a elaboração de técnicas para a definição adequada dos casos de teste baseados na análise dinâmica de especificações não foi objetivo desta tese.

Com relação à implementação, algumas limitações dizem respeito aos recursos do editor de expressões, que foi implementado de forma bastante simplificada na sua primeira versão. O editor não apresenta recursos como realce sintático das construções nem uma relação das operações utilizáveis a partir de uma expressão. Além disso, o apoio automatizado disponível através da sua interface às operações de reestruturação limita-se à execução automática de testes de regressão.

9.4 Trabalhos Futuros

Como perspectivas de trabalhos futuros decorrentes desta tese, podemos destacar:

- O desenvolvimento de técnicas complementares baseadas, por exemplo, em revisões e provas formais, não apenas para a avaliação da preservação da semântica das restrições em operações de reestruturação, mas também para outras atividades de verificação e validação de modelos com restrições especificadas em OCL. Em particular, estamos interessados em avaliar a viabilidade do desenvolvimento de técnicas de leitura específicas para apoiar a verificação de especificações OCL.
- A repetição do estudo experimental descrito no capítulo 5, com um maior número de participantes, em diferentes contextos, e de forma presencial.
- O planejamento e execução de outros estudos experimentais para avaliar, de forma mais refinada, os impactos de cada *OCL smell* e reestruturação propostas nesta tese.
- O planejamento e execução de outros estudos experimentais para avaliar os aspectos de apoio automatizado proposto para a reestruturação de modelos e restrições especificadas em OCL.

- Implementação de um conversor de expressões da versão compilada de expressões OCL para a sua sintaxe concreta, de modo a possibilitar a integração da implementação das reestruturações automáticas com a ferramenta Odyssey-PSW.
- A integração da infra-estrutura desenvolvida, como a ferramenta Odyssey-PSW e a linguagem de ação OCL Action Language, por exemplo, com abordagens de desenvolvimento baseado em transformação de modelos com geração automática de código.
- Desenvolvimento de apoio automatizado para a identificação automática de *OCL Smells,* possivelmente através da utilização de elementos da abordagem já existente para a identificação automática de construções problemáticas em modelos orientados a objetos (CORREA et al., 2000).
- A integração das propostas apresentadas nesta tese com a produção de especificações informais, como por exemplo, através de casos de uso. Alguns resultados já obtidos nesta direção estão descritos em (CORREA e WERNER, 2004b) e (CORREA e WERNER, 2004c).

Referências Bibliográficas

ACKERMANN, J., 2005, "Frequently Occurring Patterns in Behavioral Specification of Software Components". In: *Proceedings of the Conference on Component-Oriented Enterprise Applications (COEA 2005)*, pp. 41-56, Erfurt, Germany, September.

AHRENDT, W., BAAR, T., BECKERT, B., et al, 2005, "The KeY Tool", *Journal on Software and System Modeling (SoSyM)*, v. 4, n. 1, pp. 32-54.

AKEHURST, D., PATRASCOIU, O., 2004, "OCL: Implementing the Standard for Multiple Metamodels", *Electronic Notes in Theoretical Computer Science*, v. 102, n. 2, pp. 21-41.

ARNOLD, R. S., 1989, "Software Restructuring", *Proceedings of the IEEE*, v. 77, n. 4, pp. 610-617.

BAAR, T., CHIOREAN, D., CORREA, A., et al, 2005, "Tool Support for OCL and Related Formalisms - Needs and Trends". In: *Satellite Events at the MoDELS'2005 Conference - Selected Papers, Lecture Notes in Computer Science*, v. 3844, pp. 1-9, Montego Bay, Jamaica, October.

BAJEC, M., KRISPER, M., 2005, "A methodology and tool support for business rule management in organisations", *Information Systems*, v. 30, n. 6, pp. 423-443.

BALAZINSKA, M., MERLO, E., DAGENAIS, M., et al, 2000, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring". In: *Proceedings of the Working Conference on Reverse Engineering*, pp. 98-107, Brisbane, Australia, November.

BARONI, A.,ABREU, F. B., 2002, "Formalizing Object-Oriented Design Metrics upon the UML Meta-Model". In: *Simpósio Brasileiro de Engenharia de Software*, pp. 130-145, Gramado, Brasil, October.

BAUERDICK, H., GOGOLLA, M., GUTSCHE, F., 2004, "Detecting OCL Traps in the UML 2.0 Superstructure: An Experince Report". In: *<<UML>> 2004 - The Unified Modeling Language: Modeling Languages and Applications. Proceedings of the 7th International Conference, Lecture Notes in Computer Science*, v. 3273, pp. 188-196, Lisbon, Portugal, October.

BERG, M., VERHOEF, M., VIGMANS, M., 1999, "Formal Specification of an Auctioning System using VDM++". In: *Proceedings of the VDM Workshop at the World Congress on Formal Methods (FM'99)*, Toulouse, France, September.

BERRY, D. M.,KAMSTIES, E., 2004, "Ambiguity in Requirements Specification". In Leite, J. C. S. P. and Doorn, J. H. (eds), *Perspectives On Software Requirements*, 1 ed., chapter 2, Kluwer Academic Publishers.

BICARREGUI, J. C., 1997, "Formal methods into practice: case studies in the application of the B method", *IEE Proceedings in Software Engineering*, v. 144, n. 2.

BOEHM, B., 1981, *Software Engineering Economics*. 1 ed. New Jersey, Prentice Hall.

BOGER, M., STURM, T., FRAGEMANN, P., 2002, "Refactoring Browser for UML". In: *Proceedings of the International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pp. 77-81, Alghero, Sardinia, Italy, May.

BOOCH, G., 1994, *Object Oriented Analysis and Design with Applications*. 2 ed. California, Addison-Wesley.

BOOCH, G., 1999, "UML in action", *Communications of the ACM*, v. 42, pp. 26-28.

BOOCH, G., RUMBAUGH, J., JACOBSON, I., 2005, *The Unified Modeling Language User Guide*. 2 ed. Massachusetts, Addison-Wesley.

BORLAND INC., 2002, "Borland Enterprise Studio". In: http://www.borland.com/about/press/2002/estudio_for_windows.html, Accessed in 02/2002.

BREU, R., GROSU, R., HUBER, F., et al, 1998, *Systems, Views and Models of UML*. 1 ed. Heidelberg, Physica Verlag.

BRIAND, L. C., LABICHE, Y., PENTA, M., et al, 2005, "An Experimental Investigation of Formality in UML-Based Development", *IEEE Transactions on Software Engineering*, v. 31, n. 10, pp. 833-849.

BROWN, W., MALVEAU, R., MCCORMICK III, H., 1998, *Anti-patterns - Refactoring Software, Architectures and Projects in Crisis*. 1 ed., Wiley Computer Publishing.

BRUCKER, A. D.,WOLFF, B., 2002, "HOL-OCL: Experiences, Consequences and Design Choices". In: *Proceedings of the International Conference on the Unified Modeling Language: Model Engineering, Concepts and Tools*, v. 2460, pp. 196-211, Dresden, Germany, October.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., et al, 1996, *Patterns-Oriented Software Architecture: A System of Patterns*. 1 ed. England, John Wiley & Sons.

CARIOU, E., MARVIE, R., SEINTURIER, L., et al, 2004, "OCL for the Specification of Model Transformation Contracts". In: *Proceedings of the UML 2004 Workshop: OCL and Model Driven Engineering*, pp. 69-83, Lisbon, Portugal, October.

CHEESMAN, J., DANIELS, J., 2001, *UML Components: A Simple Process for Specifying Component-Based Software*. 1 ed. New Jersey, Addison Wesley.

CHIKOFSKY, E. J., CROSS, J. H., 1990, "Reverse engineering and design recovery: A taxonomy", *IEEE Software*, v. 7, n. 1, pp. 13-17.

CHIOREAN, D., BORTES, M., CORUTIU, D., 2004, "OCLE, a Tool Supporting Teaching and Learning UML and OCL, the Understanding and Using of Metamodeling, Abstraction and Design by Contract". In: *Proceedings of the 8th Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts*, pp. 1-9, Oslo, Norway, June.

CLARKE, E., GRUMBERG, O., LONG, D., 1994, "Model checking and abstraction", *ACM Transactions on Programming Languages and Systems*, v. 16, n. 5, pp. 1512-1542.

COAD, P., YOURDON, E., 1991, *Object Oriented Analysis*. 2 ed., Prentice Hall.

COOK, S., KLEPPE, A., MITCHELL, R., et al., 2002, "The Amsterdam Manifesto on OCL". In: *Clark, T. and Warmer, J. (eds), Advances in Object Modelling with the OCL*, v. 2263, *Lecture Notes in Computer Science*, Springer.

CORNÉLIO, M., CAVALCANTI, A., SAMPAIO, A., 2002, "Refactoring by Transformation", *Electronic Notes in Theoretical Computer Science*, v. 70, n. 3, pp. 641-660.

CORREA, A., WERNER, C., 2006, "Refactoring OCL Specifications", *Journal on Software and System Modeling (SoSyM), Springer-Verlag - aceito para publicação*.

CORREA, A.,WERNER, C., 2004a, "Applying Refactoring Techniques to UML/OCL Models". In: *<<UML>> 2004 - The Unified Modeling Language: Modeling Languages and Applications. Proceedings of the 7th International Conference, Lecture Notes in Computer Science*, v. 3273, pp. 173-187, Lisbon, Portugal, Octobera.

CORREA, A.,WERNER, C., 2004b, "Precise Specification and Validation of Transactional Business Software". In: *Proceedings of the International Conference on Requirements Engineering (RE'04)*, pp. 16-25, Kyoto, Japan, Septemberb.

CORREA, A.,WERNER, C., 2004c, "Specification and Validation of Transactional Business Software: An Approach Based on the Exploration of Concrete Scenarios". In: *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*, pp. 294-299, Banff, Canada, Junec.

CORREA, A., ZAVERUCHA, G., WERNER, C., 2000, "Object Oriented Design Expertise Reuse: An approach based on Heuristics, Design Patterns and Anti-patterns". In: *Proceedings ot the 6th International Conference on Software Reuse: Advances in Software Reusability, Lecture Notes in Computer Science*, v. 1844, pp. 336-352, Vienna, Austria, June.

D'SOUZA, D., WILLS, A. C., 1998, *Objects, Components and Frameworks with UML*. 1 ed., Addison Wesley.

DATE, C. J., 2000, *What Not How: The Business Rules Approach to Application Development* Massachusetts, Addison Wesley.

DEMARCO, T., 1978, *Structured Analysis and System Specification*. 1 ed., Yourdon Press.

DUCASSE, S., RIEGER, M., DEMEYER, S., 1999, "A Language Independent Approch for Detecting Duplicated Code". In: *Proceedings of the International Conference on Software Maintenance*, pp. 109-118, Oxford, England, September.

ELMSTROM, R., LARSEN, P. G., LASSEN, P. B., 1994, "The IFAD VDM-SL Toolbox: A practical approach to formal specifications", *ACM SIGPLAN Notices*, v. 29, n. 9.

EMDEN, E.,MOONEN, L., 2002, "Java Quality Assurance by Detecting Code Smells". In: *Proceedings of the Working Conference on Reverse Engineering*, pp. 97-108, Virginia, USA, October.

ERIKSSON, H. E., PENKER, M., 2000, *Business Modeling with UML*. 1 ed., John Wiley & Sons.

EVANS, A., 1998, "Reasoning with UML class diagrams". In: *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pp. 102-113, Florida, USA.

FENKAM, P., GALL, H., JAZAYERI, M., 2002, "Visual Requirements Validation: Case Study in a Corba-Supported Environment". In: *IEEE International Requirements Engineering Conference (RE'02)*, pp. 81-90, Essen, Germany, September.

FERNANDEZ-MEDINA, E.,PIATTINI, M., 2004, "Extending OCL for Secure Database Development". In: *<<UML>> 2004 - The Unified Modeling Language: Modeling Languages and Applications, Lecture Notes in Computer Science*, v. 3273, pp. 380-394, Lisbon, Portugal, October.

FEWSTER, M., GRAHAM, D., 1999, *Software Test Automation*. 1 ed., Addison-Wesley.

FINNEY, K., FENTON, N., FEDOREC, A., 1999, "Effects of structure on the comprehensibility of formal specifications", *IEE Proceedings - Software*, v. 146, n. 4, pp. 193-202.

FOWLER, M., 1999, *Refactoring: Improving the Design of Existing Programs*. 1 ed. Massachusetts, Addison-Wesley.

GAMMA, E., HELM, R., JOHNSON, R., et al, 1995, *Design Patterns: Elements of Reusable Object Oriented Software*. 1 ed. Massachusetts, Addison Wesley.

GANE, C., SARSON, T., 1979, *Structured System Analysis: Tools and Techniques*. 1 ed., Prentice Hall.

GEORGE, B., WILLIAMS, L., 2004, "A structured experiment of test-driven development", *Information and Software Technology*, v. 46, pp. 337-342.

GHEYI, R., MASSONI, T., BORBA, P., 2005, "A rigorous approach for providing model refactorings". In: *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 372-375, Long Beach, USA.

GOMAA, H., 1997, "The Impact of Prototyping on Software System Engineering", *Software Requirements Engineering*, 2 ed., Los Alamitos, California, IEEE Computer Society Press.

GORP, P. V., STENTEN, H., MENS, T., et al, 2003, "Towards Automating Source-Consistent UML Refactorings". In: *<<UML 2003>> The Unified Modeling Language: Modeling Languages and Applications. Proceedings of the Sixth International Conference, Lecture Notes in Computer Science*, pp. 144-158, San Francisco, USA, October.

GRAVELL, A., HENDERSON, P., 1996, "Executing Formal Specifications Need Not be Harmful", *IEE Software Engineering Journal*, v. 11, n. 2, pp. 104-110.

GRIWSWOLD, W. G., 1991, *Program Restructuring as an Aid to Software Maintenance*, Tese de Doutorado, University of Washington, Washington, USA.

HALL, A., 1996, "Using Formal Methods to Develop an ATC Information System", *IEEE Software*, v. 13, n. 2, pp. 66-76.

HALLE, B., 2001, *Business Rules Applied*. 1 ed. New York, John Wiley and Sons.

HAREL, D., GERY, E., 1997, "Executable Object Modeling with StateCharts", *IEEE Computer*, v. 30, n. 7, pp. 31-42.

HAZEL, D., STROOPER, P., TRAYNOR, O., 1997, "Possum: An Animator for the SUM Specification Language". In: *Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC 97/ICSC 97)*, pp. 42-51, Hong Kong, China, December.

HAZEL, D., STROOPER, P., TRAYNOR, O., 1998, "Requirements Engineering and Verification using Specification Animation". In: *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, pp. 302-305, Honolulu, Hawaii, October.

HAZELRIGG, G. A., 1999, "On the role and use of mathematical models in engineering design", *Journal of Mechanical Design*, v. 121, n. 3, pp. 336-341.

HEWITT, M. A., 1997, *PiZA: User Guide*.

HUMPHREY, W. S., 1995, *A Discipline for Software Engineering*. 1 ed., Addison Wesley.

JACOBSON, I., 1992, *Object Oriented Software Engineering: a Use Case Driven Approach*. 1 ed., Addison-Wesley.

JIA, X., 1995, "An Approach to Animating Z Specifications". In: *Proceedings of 19th Annual International Computer Software and Applications Conference*, pp. 108-113, Dallas, Texas, USA, August.

JIA, X., 1997, "A Pragmatic Approach to Formalizing Object Oriented Modeling and Development". In: *Proceedings of the 21st Annual IEEE International Computer Software and Applications Conference (COMPSAC'97)*, pp. 240-245, Washington D.C., USA, August.

JONES, C. B., 1989, *Systematic Software Development Using VDM*. 1 ed., Prentice-Hall.

KABIRA TECHNOLOGIES INC., 2006, "Kabira Action Semantics". In: http://www.kabira.com, Accessed in 01/2006.

KATAOKA, Y., ERNST, M. D., GRISWOLD, W. G., et al, 2001, "Automated Support for Program Refactoring Using Invariants". In: *Proceedings of the International Conference on Software Maintenance*, pp. 736-743, Florence, Italy.

KAZMIERCZAK, E., DART, P., STIRLING, L., et al, 2000, "Verifying Requirements through matematical modeling and animation", *International Journal of Software Engineering and Knowledge Engineering*, v. 10, n. 2, pp. 251-273.

KENNEDY CARTER LTD., 2002, "Supporting Model Driven Architecture with eXecutable UML". In: http://www.kc.com, Accessed in 02/2002.

LAITENBERGER, O.,ATKINSON, C., 1999, "Generalizing perspective-based inspection to handle object-oriented development artifacts". In: *Proceedings of the 21st Conference on Software Engineering*, pp. 494-503, Los Angeles, Estados Unidos.

LIEBERHERR, K., HOLLAND, I., 1989, "Formulations and Benefits of the Law of Demeter", *SIGPLAN Notices*, v. 24, n. 3, pp. 67-78.

LIU, S., 2004, *Formal Engineering for Industrial Software Development using the SOFL Method*. 1 ed. Heidelberg, Springer Verlag.

LOECHER, S., OCKE, S., 2004, "A Metamodel-based OCL Compiler for UML and MOF", *Electronic Notes in Theoretical Computer Science*, v. 102, n. 2, pp. 43-61.

MALDONADO, J. C.,FABBRI, S. C. P. F., 2001, "Verificação e Validação de Software". In ROCHA, A. R. C., MALDONADO, J. C., and WEBER, K. C. (eds), *Qualidade de Software - Teoria e Prática*, 1 ed., chapter 3.4, São Paulo, Prentice Hall.

MASSONI, T., GHEYI, R., BORBA, P., 2005, "Formal Refactoring for UML Class Diagrams". In: *XIX Simpósio Brasileiro de Engenharia de Software*, pp. 152-167, Uberlândia, Brasil, October.

MCCARTHY, J., HAYES, P. J., 1969, "Some philosophical problems from the standpoint of artificial intelligence", *Machine Intelligence*, pp. 463-502.

MELLOR, S., 1998, "Software Platform Independent Precise Action Specification for UML". In: *Proceedings of the International Conference on the Unified Modeling Language*, pp. 281-286, Mulhouse, France, October.

MELLOR, S., BALCER, M. J., 2002, *Executable UML: a Foundation for Model-Driven Architecture*. 1 ed. Massachusetts, Addison Wesley.

MENS, T., TOURWÉ, T., 2004, "A Survey of Software Refactoring", *IEEE Transactions on Software Engineering*, v. 30, n. 2, pp. 126-139.

MEYER, B., 1992, "Applying Design by Contract", *IEEE Computer*, v. 25, n. 10, pp. 40-51.

MILLER, T.,STROOPER, P., 2001, "Animation Can Show Only the Presence of Errors, Never Their Absence". In: *Proceedings of the 2001 Australian Software Engineering Conference (ASWEC 2001)*, pp. 76-88, Camberra, Australia, August.

NIPKOW, T., PAULSON, L. C., WENZEL, M., 2002, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. 1 ed., Springer.

OMG, 1999, "Object Management Group - Unified Modeling Language (UML) 1.3 specification". In: http://www.omg.org/cgi-bin/doc?formal/00-03-01, Accessed in 01/2006.

OMG, 2002a, "Object Management Group - MOF 1.4 specification". In: http://www.omg.org/cgi-bin/doc?formal/2002-04-03, Accessed in 01/2006a.

OMG, 2002b, "OMG-XML Metadata Interchange (XMI) Specification, v1.2". In: http://www.omg.org/cgi-bin/doc?formal/02-01-01, Accessed in 01/2006b.

OMG, 2003a, "Object Management Group - UML 2.0 OCL Specification". In: http://www.omg.org/cgi-bin/doc?ptc/2003-10-14, Accessed in 01/2006a.

OMG, 2003b, "Object Management Group - Unified Modeling Language (UML) 1.5 specification". In: http://www.omg.org/cgi-bin/doc?formal/03-03-01, Accessed in 01/2006b.

OMG, 2004a, "Object Management Group - QVT - Query/Views/Transformations, version 1.8". In: www.omg.org.

OMG, 2004b, "Object Management Group - Unified Modeling Language (UML) Superstructure Specification, version 2.0". In: http://www.omg.org/cgi-bin/doc?ptc/2004-10-02, Accessed in 2004b.

OMG, 2005, "Object Management Group - Unified Modeling Language (UML) Superstructure Specification, version 2.0". In: http://www.omg.org/cgi-bin/doc?formal/05-07-04, Accessed in 01/2006.

OPDYKE, W. F., 1992, *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*, Tese de Doutorado, University of Illnois at Urbana-Champaign, Illnois, USA.

PARNAS, D. L., 1972, "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM*, v. 15, n. 12, pp. 1053-1058.

PFLEEGER, S. H., 2001, *Software Engineering: Theory and Practice*. 2 ed. New Jersey, Prentice Hall.

PORRES, I., 2003, "Model Refactorings as Rule-Based Update Transformations". In: *<<UML 2003>> The Unified Modeling Language: Modeling Languages and Applications. Proceedings of the Sixth International Conference, Lecture Notes in Computer Science*, v. 2863, pp. 159-174, San Francisco, Estados Unidos.

PRESSMAN, R. S., 2004, *Software Engineering: a Practitioner's Approach*. 6 ed., McGraw-Hill.

RAMOS, R., SAMPAIO, A., MOTA, A., 2005, "A semantics for UML-RT active classes via mapping into CircusProc". In: *7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, pp. 99-114, Athens, Greece.

RICHTERS, M.,GOGOLLA, M., 2000, "Validating UML Models and OCL Constraints". In: *Proceedings of UML'2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, Lecture Notes in Computer Science*, v. 1939, pp. 265-277, York, England, October.

RICHTERS, M., GOGOLLA, M., 2002, "OCL: Syntax, Semantics and Tools". In: *Object Modelling with the OCL: The Rationale behind the Object Constraint Language*, v. 2263, *Lecture Notes in Computer Science*, Springer, pp. 38-63.

ROBERTS, D., 1999, *Practical Analysis for Refactoring*, Tese de Doutorado, University of Illnois at Urbana-Champaign, Illnois, USA.

ROSS, D. T., GOODENOUGH, J. B., IRVINE, C. A., 1975, "Software Engineering: Process, Principles and Goals", *IEEE Computer*, v. 8, n. 5, pp. 17-27.

RUMBAUGH, J., 1990, *Object Oriented Modeling and Design*. 1 ed., Prentice Hall.

RUMBAUGH, J., JACOBSON, I., BOOCH, G., 2004, *The Unified Modeling Language Reference Manual*. 2 ed. Massachusetts, Addison-Wesley.

SAFF, D.,ERNST, M. D., 2004, "An experimental evaluation of continuous testing during development". In: *Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pp. 76-85, Boston, USA, July.

SATPATHY, M., SNOOK, C., HARRISON, R., et al, 2001, "A Comparative Study of Formal and Informal Specifications of a Case Study from Industry". In: *Proc. IEEE/IFIP Joint Workshop Formal Specifications of Computer-Based Systems*, pp. 133-137, Washington DC, USA, April.

SCHÄFER, T., KNAPP, A., MERZ, S., 2001, "Model Checking UML State Machines and Collaborations", *Electronic Notes in Theoretical Computer Science, Elsevier Science*, v. 55, n. 3, pp. 357-369.

SENDALL, S., 2002, *Specifying Reactive System Behavior*, Tese de Doutorado, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

SHULL, F., CARVER, J., TRAVASSOS, G. H., 2001, "An Empirical Methodology for Introducing Software Processes". In: *Proceedings of the Joint 8th European Software Engineering Symposium and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 288-296, Vienna, Austria, September.

SIMON, F., STEINBRÜCNER, F., LEWERENTZ, C., 2001, "Metrics based refactoring". In: *Proceedings of the European Conference on Software Maintenance and Reengineering*, pp. 30-38, Lisbon, Portugal.

SNOOK, C. F.,HARRISON, R., 2001, "Experimental Comparison of the Comprehensibility of a Z Specification and its Implementation". In: *Proceedings of the Conference on Empirical Assesment in Software Engineering - EASE 01*, Keele University, England, April.

SOMMERVILLE, I., 1992, *Software Engineering*. 4 ed., Addison Wesley.

SOMMERVILLE, I., SAWYER, P., 1997, *Requirements Engineering: a good practice guide*. 1 ed. England, John Wiley & Sons.

SUNYÉ, G., POLLET, D., LETRAON, Y., et al, 2001, "Refactoring UML models". In: *<<UML 2001>> The Unified Modeling Language: Modeling Languages, Concepts and Tools. Proceedings of the Forth International Conference, Lecture Notes in Computer Science*, v. 2185, pp. 134-148, Toronto, Canada, October.

TOKUDA, L., BATORY, D., 2001, "Evolving Object-Oriented Designs with Refactorings", *Automated Software Engineering*, v. 8, n. 1, pp. 89-120.

TORO, A. D., 2000, *Un Entorno Metodológico de Ingeniería de Requisitos para Sistemas de Información*, Tese de Doutorado, Universidad de Sevilla, Sevilla, Spain.

TOURWÉ, T.,MENS, T., 2003, "Identifying Refactoring Opportunities using Logic Meta Programming". In: *Proceedings of the European Conference on Software Maintenance and Re-engineering*, pp. 91-100, Benevento, Italy.

TOVAL, A., REQUENA, V., FERNANDEZ, J. L., 2003, "Emerging OCL Tools", *Journal on Software and System Modeling (SoSyM)*, v. 2, n. 4, pp. 248-261.

TRAVASSOS, G. H., SHULL, F., CARVER, J., 2001, "Working with UML: A software design process based on inspections for the unified modeling language", *Advances in Computers*, v. 54, n. 1, pp. 35-97.

TRAVASSOS, G. H., SHULL, F., FREDERICKS, M., et al, 1999, "Detecting Defects in Object Oriented Designs: Using Reading Techniques to increase Software Quality", *ACM SIGPLAN Notices*, v. 34, n. 10, pp. 47-56.

VAZIRI, M.,JACKSON, D., 2002, "Some Shortcomings of OCL, the Object Constraint Language of UML". In: *Proceedings of the Technology of Object Oriented Languages and Systems Conference (TOOLS USA 34)*, pp. 555-562, Santa Barbara, California, USA, July.

WARMER, J., KLEPPE, A., 2003, *The Object Constraint Language. Getting Your Models Ready for MDA*. 2 ed. Reading, Mass, Addison Wesley.

WEST, M. M., EAGLESTONE, B., 1992, "Software Development: two approaches to animation of Z specifications using Prolog", *Software Engineering Journal*, v. 7, n. 4.

WOHLIN, C., RUNESON, P., HÖST, M., et al, 2000, *Experimentation in Software Engineering: An Introduction*. 1 ed. Massachusetts, Kluwer Academic Publishers.

WOODCOCK, J., DAVIES, J., 1996, *Using Z. Specification, Refinement and Proof*. 1 ed., Prentice Hall.

YOURDON, E., 1989, *Modern Structured Analysis*. 1 ed., Prentice Hall.