VINÍCIUS COELHO DE ALMEIDA

USO DA LINGUAGEM OCL NO CONTEXTO DE DIAGRAMAS DE CLASSE DA UML E PROGRAMAS EM JAVA

Belo Horizonte
05 de julho de 2006

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

USO DA LINGUAGEM OCL NO CONTEXTO DE DIAGRAMAS DE CLASSE DA UML E PROGRAMAS EM JAVA

Proposta de dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

VINÍCIUS COELHO DE ALMEIDA

Belo Horizonte
05 de julho de 2006

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Uso da linguagem OCL no contexto de diagramas de classe da UML e programas em Java

VINÍCIUS COELHO DE ALMEIDA

Proposta de dissertação defendida e aprovada pela banca examinadora constituída por:

Ph. D. Rodolfo Sérgio Ferreira Resende – Orientador
Universidade Federal de Minas Gerais

Docteur Antônio Otávio Fernandes – Co-orientador
Universidade Federal de Minas Gerais

Ph. D. Roberto da Silva Bigonha
Universidade Federal de Minas Gerais

Doutor Mark Alan Junho Song
Pontifícia Universidade Católica de Minas Gerais

Belo Horizonte, 05 de julho de 2006

Resumo

Neste trabalho, apresentamos o uso de OCL na engenharia a frente de diagramas de classe UML para código Java, e seu possível uso no processo de engenharia reversa nesse contexto. Foi feito um levantamento do estado-da-arte das ferramentas utilizadas, em artigos acadêmicos, assim como foram coletados problemas existentes durante a engenharia a frente e reversa. Também identificamos mapeamentos entre tipos e operações de ambas as linguagens. Por fim, nossas conclusões sobre a viabilidade do uso de OCL durante o processo de desenvolvimento e proposta de trabalhos futuros finalizam a dissertação.

Abstract

In this work, we present OCL usage in forward engineering of UML class diagrams to Java code, as well as a discussion of its potential usage during reverse engineering in this context. The state-of-art of the related tools in the academic literature, and existing problems on forward and reverse engineering were surveyed. Also, we have identified mappings among types and operations from both languages. Finally, our conclusions about OCL usage viability on development process and proposal of future works finalize the dissertation.

Para meus pais, Jarbas (in memoriam) e Cleides, e meus irmãos, Júnio e Suellen.

Agradecimentos

Agradeço aos meus orientadores, os professores Rodolfo Sérgio Ferreira Resende e Antônio Otávio Fernandes, o apoio e esclarecimentos, sem os quais essa pesquisa não seria possível. Como também ao doutorando Pedro de Alcântara dos Santos Neto, que contribuiu com seus conhecimentos para o enriquecimento do trabalho.

Ao *Synergia - Laboratório de Engenharia de Software e Sistemas*, agradeço o suporte durante a pesquisa.

À turma da *Grad001* e do LECOM, o companheirismo de sempre.

À professora Karin Birgit e seu grupo de dança de salão, *Anatomia da Dança*, que tornaram essa caminhada menos árdua.

A meus amigos, que me suportam e me *suportam* ao longo desses anos, cujo apoio é uma constante e uma certeza nos momentos difíceis.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Processos de desenvolvimento de software, tipicamente, envolvem: a identificação do domínio do problema por meio de *modelos de análise*; a definição da arquitetura da solução, utilizando *modelos de desenho* (design models); e a implementação propriamente dita da solução, em uma linguagem de programação qualquer (*modelos de implementação*) [JBR98]. Neste contexto, a modelagem é um dos recursos utilizados para auxiliar no desenvolvimento e manutenção de softwares.

A UML é uma linguagem de modelagem largamente utilizada, fornecendo um conjunto de convenções diagramáticas que são usadas para auxiliar a esboçar e a documentar sistemas de software. A versão 1.0 da UML foi bastante criticada por ser definida com muito pouco rigor. Sua definição foi criada utilizando apenas os elementos gráficos da própria UML, por meio de um *meta-modelo MOF* (Meta-Object Facility). Essa abordagem permitia uma grande ambigüidade na interpretação dos modelos. Um primeiro passo em direção a uma maior precisão da especificação UML foi a criação da linguagem textual *OCL* (Object Constraint Language) [OCL03, WK03], baseada em uma linguagem desenvolvida por um grupo de pesquisa da IBM para definição de restrições em modelos. Ela é baseada em teoria de conjuntos e lógica de predicados, sendo que sua semântica formal encontra-se definida na tese de Richters [Ric02]. OCL suporta a expressão de restrições em invariantes, pré-condições e pós-condições. A princípio, ela foi utilizada unicamente para a definição de *regras de boa formação* (well-formedness rules) das especificações dos meta-modelos da UML e da própria MOF. Entretanto, durante o desenvolvimento de software, há propriedades e restrições que são muito complexas ou impossíveis de serem expressadas adequadamente em um diagrama, utilizando apenas representação iconográfica. O uso de linguagem natural para descrever essas restrições resolve parcialmente o problema, pois sua imprecisão pode dificultar a automatização do processamento do modelo. Sendo adotada como parte integrante da UML 1.1, a OCL foi disponibilizada para a comunidade de modelagem de software.

Atualmente, a OCL tem sido usada em diferentes contextos e com diferentes propósitos no desenvolvimento de software. Devos e Steegmans [DS05], além de Penker e Eriksson [PE00], utilizam OCL na modelagem de *regras de negócio*. Um analista observa e registra em um *modelo de negócio*, de modo preciso: as regras da legislação vigente, da organização ou impostas pelo negócio, dentre outras. Varella e outros [VPH$^+$04] utilizam as regras de negócio expressas em OCL para detectar eventos em bancos de dados que possam violá-las.

Na especificação da UML 2.0 [UML05, UML04], expressões OCL são utilizadas para dar maior precisão às regras de boa formação dos conceitos apresentados. Com isso, Bauerdick e outros [BGG04] identificaram automaticamente inconsistências entre as expressões OCL e a estrutura de classes da especificação, por meio de ferramenta apresentada pelos autores. A partir de *diagramas de classe* de um modelo UML, anotados com restrições OCL, Richters e Gogolla [RG03] verificaram a ocorrência de violações dessas restrições na implementação respectiva em Java.

Lavazza e Barresi [LB05] propõem uma ferramenta para suportar a criação de planos de medição de processos que utilizem a metodologia *GQM* (Goal/Question/Metrics). Em sua abordagem, métricas complexas, que não possam ser representadas diretamente por um atributo do modelo do processo, são representadas como operações, cuja semântica é descrita em OCL.

Em ferramenta de *refabricação* (refactoring) desenvolvida por Gorp e outros [GSMD03], os efeitos das refatorações realizadas são descritos utilizando contratos em OCL, o que possibilita a refabricação de desenhos (designs) independentemente da linguagem de programação subjacente ao modelo. De modo similar, Correa e Werner [CW04] utilizam técnicas de refabricação para melhorar a inteligibilidade de diagramas de classe UML anotados com OCL.

Mak e outros [MCL04], além de Zdun e Avgeriou [ZA05], utilizaram expressões OCL para modelar precisamente *mecanismos de desenho* (design patterns) [GHJV94]. Segundo os autores, com essas expressões, foi possível identificar automaticamente mecanismos de desenho presentes em um dado diagrama de classe UML, dentre outros benefícios.

Soundarajan e Fridella [SF99], bem como Gurunath [Gur04], utilizam OCL na modelagem de exceções em UML, descrevendo para o desenvolvedor as condições necessárias para que essas exceções sejam executadas.

Para auxiliar na compreensão de softwares, Antoniol e outros [APM03] propõem um interpretador OCL para a realização de consultas em *AST*s (Abstract Syntax Trees) do código do software sendo analisado. Também com o objetivo de utilizar OCL para navegação, Sakr e Gaafar [GS04] desenvolveram uma ferramenta que permite a execução de consultas em modelos UML, utilizando a OCL de maneira semelhante ao uso que é feito com *XQuery* em documentos XML. Similarmente, Siikarla e outros [SPS04] implementaram uma ferramenta para a realização de consultas em OCL sobre um modelo UML.

Hussmann e Zschaler [HZ04] sugerem o uso de OCL como linguagem de programação de contratos para a descrição de componentes de software independentes de plataforma.

Segundo Ol'khovich e Koznov [OK03], OCL pode ser usada na verificação automática da integridade e consistência de modelos UML. Por exemplo, pode-se verificar a correspondência dos valores dos atributos de objetos nos diagramas de colaboração em relação às restrições das classes correspondentes, presentes nos diagramas de classes. Além disso, sugerem que o uso de especificações OCL possa representar uma fonte de informação adicional para a realização de *engenharia direta* (forward engineering). Isso de fato ocorre. Diferentes ferramentas foram criadas por diversos autores para a utilização de OCL em diagramas de classe UML [ABB$^+$05, AP04, BAMF04, BDL04, HDF00, VS02]. Um objetivo comum entre esses autores é a posterior geração de código que verifique se as restrições modeladas em OCL são respeitadas pela implementação respectiva.

Em estudo empírico realizado por Briand e outros [BLS03], mostrou-se que o uso de asserções de

contratos no código-fonte pôde detectar um percentual significativo de falhas. Além disso, os autores concluíram que essas asserções podem ser usadas para reduzir muito o esforço de localização de defeitos após a detecção de uma falha, mesmo que esses contratos sejam incompletos. Frente a esses resultados, ambicionávamos a implementação de uma ferramenta relacionada ao contexto do estado-da-arte do uso de OCL na engenharia direta de UML para Java, assim como na *engenharia reversa* de Java para UML. Entretanto, diferentes obstáculos se apresentaram, impedindo essa implementação (apresentados na Seção 4.15). Neste trabalho, nossa principal contribuição é apresentar o uso de OCL na engenharia direta de diagramas de classe UML para código Java e no possível uso de OCL no processo de engenharia reversa de código Java para diagramas de classe UML.

Os processos de engenharia direta e engenharia reversa possuem características distintas do processo de *engenharia de ida e volta* (round-trip engineering). A engenharia direta e a engenharia reversa transformam um ou mais artefatos em outros artefatos, onde quaisquer informações presentes nos artefatos-alvo são ignoradas: cria-se um novo artefato, removendo a versão anterior. Na engenharia de ida e volta, a informação presente no artefato-alvo é preservada; a intenção é reconciliar (reconcile) modelos, e não apenas transformá-los em uma dada direção [SK04].

A ênfase da análise desta dissertação é em termos de artigos acadêmicos, uma vez que praticamente todas as ferramentas comerciais mais usuais não suportam OCL. Não houve a preocupação em se analisar ferramentas comerciais ou diferenças de suporte de OCL entre as diferentes versões dessas ferramentas. Entretanto, isso não implica que o uso de OCL seja irrelevante comercialmente, dado que algumas companhias fazem, por exemplo, plug-ins que suportam o uso de OCL nas ferramentas *Rational Rose*[1] e *Eclipse*[2], o que, de certa forma, evidencia interesse comercial pela tecnologia.

A fim de facilitar a compreensão do texto, foram utilizadas as seguintes convenções tipográficas:

- "Entre aspas": Substantivação de termos no texto.
- Sans serif: Termos em língua estrangeira e desdobramento de siglas, como por exemplo, de nomes de ferramentas ou metodologias. Ao traduzir alguns dos conceitos, será mostrado o termo original, em sua primeira ocorrência neste trabalho, entre parênteses. O significado das siglas também será mostrado entre parênteses.
- *Enfatizado (itálico)*: Ferramentas, metodologias estudadas e termos relevantes ao contexto onde foram mencionados.
- `Tipo fixo`: Operações, métodos, operadores, campos, atributos, parâmetros e argumentos mencionados em exemplos e/ou diagramas.
- 'Entre apóstrofes': Multiplicidade de associações.

Essa dissertação está organizada da seguinte maneira: no Capítulo 2 são apresentadas a terminologia usada na dissertação, bem como uma introdução à modelagem de software e à linguagem OCL; no Capítulo 3, são apresentadas as abordagens levantadas que utilizam OCL na engenharia direta de software; no Capítulo 4, são mostradas as abordagens que tratam da engenharia reversa de código Java para diagramas de classe UML; por fim, o Capítulo 5 conclui a dissertação.

[1] Add-in *Oclarity*, da empresa EmPowerTec, disponível em: http://www.empowertec.de

[2] Plug-in *Octopus*, disponível em: http://www.klasse.nl/ocl/

Capítulo 2

Modelagem e OCL

2.1 Terminologia utilizada

Esta seção apresenta uma terminologia comum de referência para toda a dissertação. Seu objetivo é tornar homogênea a apresentação das informações dos diferentes trabalhos analisados. Isso mostrou-se necessário dadas as diferentes nomenclaturas utilizadas pelos autores para representar as mesmas idéias. Alguns dos termos são tipicamente usados no contexto de modelos UML, enquanto outros são mais utilizados num contexto Java (por exemplo, atributos em UML, campos em Java). Quando se julgar necessário, esse contexto será explicitado. As definições a seguir foram obtidas (e adaptadas) principalmente dos trabalhos de: Rumbaugh e outros [RJB05], Jacobson e outros [JBR98] e Meyer [Mey97].

Fonte, código-fonte: Descrição de um software, contida em arquivos de texto, que é escrita por desenvolvedores, sendo lida e entendida por um compilador. Em Java, são os arquivos com extensão ".java".

Executável, código executável, bytecode, arquivo de classe: Descrição de um software, em arquivos binários, que pode ser executada por um computador. Em Java, tipicamente esses arquivos não são executados diretamente, sendo necessário o uso de uma *máquina virtual* para interpretá-los. São os arquivos com extensão ".class".

Software, programa, aplicação: Coleção de unidades conectadas e organizadas para o cumprimento de um propósito. Um software pode ser descrito por um ou mais modelos, possivelmente a partir de diferentes pontos-de-vista. São todos os artefatos necessários para representarem um software em forma legível para engenheiros de software (por exemplo, modelos de análise, código-fonte, especificações de requisitos), bem como para ferramentas, compiladores e computadores (por exemplo, código-fonte, executáveis).

Modelo: Artefato que representa uma abstração de um software, especificando-o a partir de um dado ponto-de-vista e em um certo nível de abstração.

Artefato: Qualquer tipo de informação criada, produzida, alterada ou usada por engenheiros de software durante o desenvolvimento de um sistema (por exemplo, código-fonte, código executável, especificação de requisitos, modelo de desenho). É a especificação de um elemento físico de informação que é usado ou produzido por um processo de desenvolvimento de software. Tipicamente, são arquivos ou documentos.

Diagrama de classe: É uma apresentação gráfica da visão estática do modelo (que mostra sua estrutura estática), mostrando uma dada coleção de elementos do modelo, declarativos e estáticos (por exemplo, classes e tipos), seu conteúdo e relacionamentos.

Arquitetura: Estrutura organizacional de um sistema, incluindo sua decomposição em partes, sua conectividade, mecanismos de interação e os princípios norteadores que informam o desenho de um software.

Associação: Relacionamento estrutural que descreve uma conexão entre objetos; relacionamento semântico entre dois ou mais classificadores (por exemplo, classes) que envolve conexões entre suas instâncias.

Associação simples: Associação sem adornos.

Agregação: Uma forma de associação que especifica um relacionamento todo-parte entre um agregado (aggregate), o todo, e uma de suas partes constituintes.

Composição: Uma forma mais forte de agregação na qual uma parte pode apenas pertencer a apenas um composto (composite), o todo, de cada vez. Partes cujas multiplicidades não estejam fixadas podem ser criadas após a criação do composto, mas, uma vez criadas, são destruídas junto com a destruição do composto.

Realização entre classe e interface, implementação de interface: Relacionamento entre uma especificação e sua implementação; uma indicação de herança de comportamento sem herança de estrutura.

Herança: Mecanismo por meio do qual classes ou interfaces mais específicas incorporam a estrutura (no caso de classes) e o comportamento (no caso de ambas) de classes ou interfaces mais gerais, respectivamente.

Dependência entre classes: Relacionamento entre dois classificadores (por exemplo, uma classe) no qual a mudança em um deles (o provedor) pode afetar ou fornecer informação necessária para outro (o cliente). Por exemplo, a dependência de permissão («`permit`»), que possibilita ao cliente usar o conteúdo do provedor independentemente de declarações de visibilidade de atributos e operações desse provedor.

Classificador: Elemento de modelo que descreve características estruturais e comportamentais. Seus tipos incluem: associação, classe e interface, sendo as classes seu tipo mais comumente usado.

Classe: Descritor para um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e comportamento.

Interface: Declaração de um conjunto coerente de propriedades e obrigações públicas; um contrato entre provedores de serviços e seus consumidores.

Instância, objeto: Uma entidade individual com sua própria identidade e valor, cuja forma e comportamento são especificados por um classificador (por exemplo, uma classe). Possui fronteira e identidade bem-definidas, encapsulando estado e comportamento. Uma estrutura de dados de tempo de execução constituída de zero ou mais campos. Representação em computador de um objeto abstrato. Todo objeto é uma instância de alguma classe.

Atributo: Descrição de um elemento nomeado, de um tipo especificado em uma classe; cada objeto da classe contém, separadamente, um valor do tipo. Contexto: diagrama de classe (vide *Campo*).

Campo: Um dos valores que constituem um objeto. Contexto: código Java (vide *Atributo*).

Operação: Especificação de uma transformação ou consulta que um objeto possa ser solicitado a executar. Contexto: diagrama de classe (vide *Método*).

Método: Implementação de uma operação. Ela especifica o algoritmo que produz os resultados de uma operação. Contexto: código Java (vide *Operação*).

Pacote: Mecanismo de propósito geral para a organização de elementos em grupos, estabelecendo a posse sobre seus elementos constituintes e fornecendo um espaço de nomes (namespace) para referenciar esses elementos.

Desenvolvedor: Indivíduo responsável por executar atividades pertinentes ao desenvolvimento de software.

Mantenedor (maintainer): Desenvolvedor que atua durante as etapas de pós-implantação de software, em sua manutenção.

Desenhista de software (software designer): Desenvolvedor que cria os modelos de um software (por exemplo, de análise, de desenho).

2.2 Modelagem e desenvolvimento de software

Segundo Bèzivin [Bèz05], "a Ciência da Computação pode ser descrita como a ciência da construção de modelos de software, onde um modelo é uma representação de um sistema". Modelos de software são divididos em duas categorias: estáticos versus dinâmicos, prescritivos versus descritivos. Na primeira categoria, os modelos são classificados conforme sua mudança com o passar do tempo (a maioria dos modelos é estática, enquanto que a maioria dos sistema é dinâmica). A segunda categoria classifica os modelos em relação a como são usados: um modelo é *prescritivo* se determina como um sistema deve ser construído, tipicamente em engenharia direta; ou é *descritivo*, caso o modelo seja construído a partir da observação de um sistema em operação, tipicamente em engenharia reversa para redocumentar sistemas existentes.

Segundo Ol'khovich e Koznov [OK03], a modelagem é usada como meio de comunicação entre pessoas, admitindo a existência de diagramas baseados em uma notação comumente usada, que pode ser entendida intuitivamente. Para Chung e Lee [CL01], é um processo para visualizar, especificar, construir e documentar artefatos em diagramas padronizados, facilitando a comunicação entre as diferentes partes interessadas de um projeto de desenvolvimento de software. Conforme lembrado por Henderson-Sellers [HS05], "há não muito tempo, eram usados *fluxogramas* (flow charts), um tipo primitivo de linguagem de modelagem de software, que abstrai detalhes sobre a tecnologia de programação, utilizando uma notação que expressa o fluxo de controle de um algoritmo".

Gogolla e outros [GBR05], Zdun e Avgeriou [ZA05], Henderson-Sellers [HS05], Briand e outros [BDL04], dentre outros autores, afirmam que a *UML* (Unified Modeling Language) [UML05, UML04, RJB05, Fow05] é a linguagem de modelagem de facto da indústria de desenvolvimento de software na atualidade. A UML fornece um conjunto de convenções diagramáticas que são usadas para auxiliar a esboçar e documentar sistemas de software, sendo um grande sucesso, segundo Henderson-Sellers [HS05]. Ainda segundo esse autor, "de maneira crescente nesses últimos poucos anos, algumas organizações têm usado UML para impulsionar a geração de código e outros artefatos de software (...), principalmente, a partir de diagramas de classe, os quais são usados na

geração de interfaces para os vários componentes de uma aplicação distribuída". A UML fornece vários pontos-de-vista em alto nível por meio de uma série de diagramas. Isso dá aos usuários a habilidade de especificar sistemas utilizando qualquer combinação conveniente desses diagramas. Em sua versão 2.0, são oferecidos 13 diagramas: de atividade, de classe, de comunicação, de componentes, de estruturas compostas, de implantação, de panorama de interação, de objeto, de pacote, de máquina de estado, de seqüência, de temporização e de caso de uso [RJB05]. A UML não é específica para uma linguagem de implementação particular [CKvKR04]. Ferramentas que suportam a UML encontram-se disponíveis em pacotes comerciais (por exemplo, Borland Together, Rational Rose) e não-comerciais (por exemplo, ArgoUML). Na maioria dessas ferramentas, o formato *XMI* (eXtensible Markup Language Metadata Interchange) é geralmente suportado, sendo utilizado como um meio textual de armazenamento dos diagramas.

No desenvolvimento de software sob a égide de um processo como o *Processo Unificado*, parte-se da construção de modelos informais, que a cada iteração são refinados, criando-se modelos mais precisos (i.e., mais facilmente interpretáveis por um computador). Inicia-se o processo, capturando os requisitos do cliente em um *modelo de casos de uso*; esses requisitos são analisados, produzindo-se seu *modelo de análise*; a estrutura estática do sistema é desenhada para atender aos casos de uso levantados, criando-se o *modelo de desenho*; por fim, o sistema é implementado em um *modelo de implementação*, que inclui os componentes (código-fonte e executáveis) e o mapeamento entre esses componentes e as classes de desenho respectivas [JBR98]. Distinções mais sistemáticas e claras entre o modelo de desenho e o modelo de implementação, feitas pela comunidade da UML, são um fenômeno recente. Por exemplo, por meio da *MDA* (Model-Driven Architecture), é feita a separação entre *PIM* (Platform Independent Model) e *PSM* (Platform Specific Model). Este último é usado na geração subseqüente de esquemas, código, especificações de implantação e outros artefatos [Bèz05, HS05]. Nosso trabalho envolve modelos mais precisos, como os modelos de desenho e de implementação, em relação aos elementos UML a serem usados em engenharia direta e em engenharia reversa.

2.3 Importância de diagramas de classe

Dentre os diagramas da UML, os *diagramas de classe* são os diagramas mais importantes de modelagem de objetos da UML [RJB05]. Eles mostram a estrutura estática dos objetos em um sistema. Segundo Kollmann e outros, "são geralmente considerados os diagramas mais bem empregados e os mais bem compreendidos dentre os diagramas da UML" [KSS+02]. Eles descrevem a estrutura estática dos sistemas, em um nível mais alto de abstração do que o código-fonte, mostrando classes, interfaces e seus relacionamentos. Guéhéneuc [Gué04c] afirma que "diagramas de classe UML são um recurso valioso para auxiliar engenheiros de software – sejam desenvolvedores, sejam mantenedores – a entender as arquiteturas, comportamentos, escolhas de desenho e implementação de programas". Desenvolvedores de software utilizam esses diagramas para descrever a arquitetura de softwares orientados por objeto durante desenvolvimento, auxiliando-os na abstração de detalhes de implementação. Além disso, diagramas de classe auxiliam mantenedores de software a entender a arquitetura dos softwares e a localizar trechos que exijam modificações durante sua manutenção.

2.4 O que é OCL?

Apesar do poder de expressão de linguagens gráficas e visuais de modelagem como a UML, há propriedades e restrições de softwares que são muito complexas ou impossíveis de serem expressadas em um diagrama adequadamente por meio de uma representação exclusivamente iconográfica. Mesmo os mecanismos de extensão da UML – *estereótipos* (stereotypes), *valores etiquetados* (tagged values) e *restrições* (constraints) pré-definidas – podem ser insuficientes nesse sentido. Por exemplo, seja o diagrama de classe de uma aplicação fictícia representado na Figura 2.1, extraído do trabalho de Warmer e Kleppe [WK03].

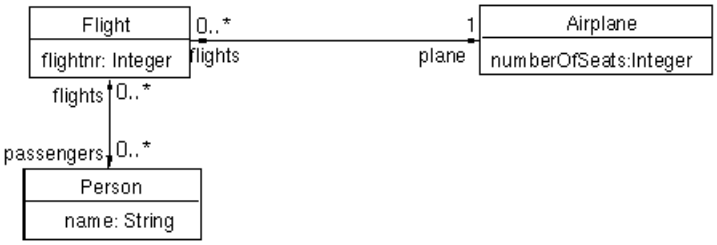

Figura 2.1: Diagrama-exemplo do uso de OCL: máximo de passageiros por vôo

Como um desenhista de software poderia representar neste diagrama, para um implementador, que o número máximo de passageiros (papel `passengers`) em um vôo (classe `Flight`) é limitado pela capacidade do avião respectivo (classe `Airplane`, atributo `numberOfSeats`)? Multiplicidades podem ser representadas utilizando puramente UML, mas neste caso isso não é suficiente. A multiplicidade '0..*' da associação entre as classes `Flight` e `Person`, próximo à `Person`, não é capaz de refletir essa restrição, dado que o número de lugares dos aviões pode variar para cada avião. Características como esta não podem ser expressas utilizando apenas multiplicidades, estereótipos ou valores etiquetados, devendo ser registradas nos modelos utilizando restrições em linguagem natural. Entretanto, Ol'khovich e Koznov afirmam que, quando especificações UML devem ser submetidas a processamento automatizado, tais como geração de código, exige-se um maior grau de precisão quanto às informações fornecidas [OK03]. Desde a versão 1.1, a UML é amparada pela OCL, uma linguagem puramente textual para descrição de restrições. Ela é capaz de especificar informações adicionais em diagramas UML as quais não podem ser descritas graficamente. Ela é baseada em lógica de predicados e teoria de conjuntos, suportando a expressão de restrições que refletem o paradigma de Projeto por Contrato (Design by Contract), de Meyer [Mey97]: *invariantes* (propriedades que caracterizam uma classe como um todo, sendo verdadeiras em todas as suas instâncias), *pré-condições* (condições que devem ser satisfeitas para que uma dada operação seja executada com sucesso) e *pós-condições* (condições garantidas por uma operação após o término de sua execução bem-sucedida). Pré-condições e pós-condições representam as duas partes de um contrato: se a instância cliente de uma dada operação garante que as pré-condições dessa operação sejam satisfeitas, a instância fornecedora da operação garante que a pós-condição será verdadeira, terminada a execução da operação.

OCL é uma linguagem fortemente tipada. Seus tipos encontram-se organizados em uma hierarquia, conforme representado na Figura 2.2, extraída de sua especificação [OCL03].

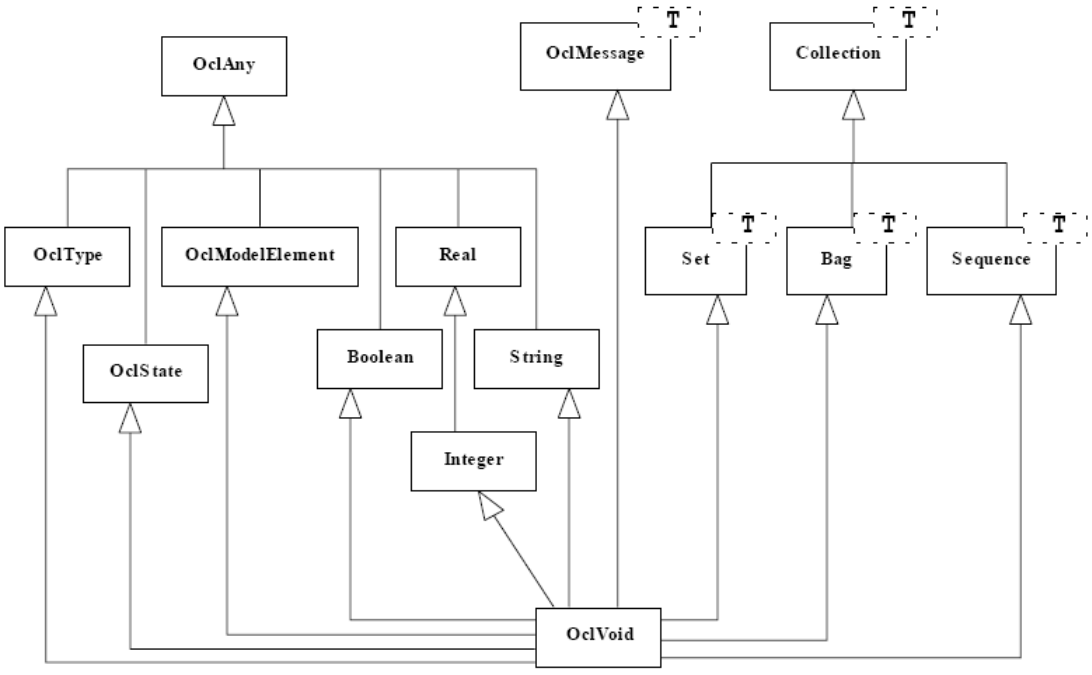

Figura 2.2: Tipos definidos na biblioteca padrão da OCL

Os principais tipos de OCL são divididos em: tipos primitivos, tipos coleção e tuplas. Os tipos primitivos são: `Boolean`, `Integer`, `Real` e `String`. Os tipos coleção são: `Collection` (aglomerado qualquer de elementos do mesmo tipo, duplicados ou não, ordenados ou não; coleção), `Set` (conjunto de elementos do mesmo tipo, não-ordenados, sem duplicatas; conjunto matemático), `Bag` (como `Set`, mas permite duplicatas; multi-conjunto), `OrderedSet` (como `Set`, mas seus elementos possuem um índice interno de ordenação; conjunto ordenado) e `Sequence` (como `Bag`, mas seus elementos possuem índice interno de ordenação). Um esquema é mostrado na Tabela 2.1. Já as tuplas (tipo

Tabela 2.1: Tipos coleção de OCL

	Sem repetição	Com repetição
Sem ordenação	Set	Bag
Com ordenação	OrderedSet	Sequence

`TupleType`) são composições de valores, cujos tipos desses valores podem ser diferentes, como por exemplo: `Tuple{name='John', age=10}`[1].

Expressões em OCL não possuem efeitos colaterais: apenas retornam um valor, não alterando nenhuma informação do modelo. Sua avaliação é instantânea – os estados dos objetos não podem ser alterados durante essa avaliação. Além disso, utilizando expressões OCL, é possível a navegação entre as classes de um modelo UML, a partir de uma classe-base, que fornece o contexto da expressão. Para acessar um atributo ou operação, navega-se pelas associações utilizando o operador "."; para acessar propriedades de uma coleção – que pode ser definida pelas instâncias de uma classe, ou

[1]Não encontramos explicação sobre os tipos `OrderedSet` e `TupleType` não estarem representados na hierarquia de tipos da especificação da linguagem.

pelas instâncias ligadas (linked) a uma dada instância de uma classe –, utiliza-se o operador "->". Para ilustrar isso, uma solução para a restrição do exemplo da Figura 2.1 pode ser formulada em OCL:

```
context Flight inv passageirosPorVoo:
  self.passengers->size() <= self.plane.numberOfSeats
```

O contexto de onde parte a navegação é fornecido (classe `Flight`); a restrição é um invariante (palavra-chave `inv`) nomeado (`passageirosPorVoo`). Para a avaliação do lado esquerdo da inequação, navega-se do contexto (`self`) até o alvo da associação (papel `passengers`); como sua multiplicidade é '*', esse sentido da associação aponta para uma coleção de elementos (neste caso, um conjunto); para acessar uma das propriedades dessa coleção, utiliza-se o operador "`->`". A operação executada (`size()`) retorna o número de elementos dentro da coleção, na forma de um inteiro (tipo OCL `Integer`). Para a avaliação do lado direito, navega-se do contexto (`self`) em direção ao alvo da outra associação (papel `plane`); como sua multiplicidade é '1', esse sentido da associação aponta para um único elemento (operador "`.`"), cujo atributo pode ser recuperado (`numberOfSeats`), retornando-se um inteiro. Por fim, efetua-se a comparação entre os inteiros obtidos em ambos os lados da inequação (OCL não especifica uma ordem para avaliação; apenas para fins de explicação do exemplo, avaliou-se o lado esquerdo primeiro).

Obviamente, esta é uma expressão simples, mas existem outras situações mais complexas onde OCL se mostra útil. Considere o exemplo a seguir, onde são mostradas as assinaturas das operações de uma classe chamada `Pilha`[2].

```
class Pilha {
    Object desempilhar();
    void empilhar(Object objeto);
    Object topo();
    boolean estahVazia();
}
```

Diferente do exemplo anterior, as restrições são impostas nas operações da classe. Em linguagem natural, as restrições são as seguintes:

- a operação `empilhar()` acrescenta um objeto (`java.lang.Object`) ao topo da pilha;
- a operação `topo()` apenas retorna o objeto do topo da pilha, não o removendo;
- a operação `desempilhar()` remove o objeto do topo da pilha, retornando-o;
- caso a pilha esteja vazia, não é possível retornar nem desempilhar elemento algum.

Vale ressaltar que esse contrato não visa representar exaustivamente o tipo abstrato de dados de mesmo nome, nem expressar a semântica das operações. Visa, apenas, apresentar um exemplo de utilização de OCL expressando restrições na descrição mais precisa de contratos (para uma discussão mais localizada sobre a documentação de contratos, vide Seção 4.15, iniciativa *Documentação de contratos de classes e interfaces de coleção*). Uma possível representação em OCL dessas restrições seria:

[2] Adaptado de http://www.empowertec.de/ocl/example3.htm, acessado em 21/12/2005.

```
context Pilha::empilhar(objeto: Object)
   post objetoEmpilhadoNoTopo: self.topo() = objeto
context Pilha::topo(): Object
   pre naoVazia: not self->estahVazia()
   post retornaTopo: result = self->first()
context Pilha::desempilhar(): Object
   pre naoVazia: not self->estahVazia()
   post elementoRemovido: self->excludes(result)
   post elementoRetornado: result = self@pre->topo()
```

Para a operação `empilhar`, é fornecida uma pós-condição (palavra-chave `post`) na qual verifica-se que, ao fim de sua execução, o resultado da operação `topo` seja o mesmo objeto empilhado. Para a operação `topo`, há a pré-condição (palavra-chave `pre`) que a pilha não esteja vazia; também possui a pós-condição que o elemento a ser retornado deve ser o primeiro. Por fim, a operação `desempilhar` possui uma pré-condição que exige que a pilha não esteja vazia, e duas pós-condições: APÓS o desempilhamento, o objeto não deve estar presente na pilha (`self->excludes(result)`); o resultado da operação (palavra-chave `result`) deverá ser o mesmo elemento que estava no topo da pilha ANTES de sua execução (`self@pre.topo()`). O uso da palavra-chave `self` serve para indicar que os atributos e operações sendo analisados são de uma instância da própria classe informada no contexto da operação (i.e., `Pilha`). Seu uso é opcional, exceto nos casos onde se refere a valores anteriores à execução de uma operação, nos quais é necessário utilizar o operador `@pre`.

Em outro exemplo, seja o diagrama de classe representado na Figura 2.3, adaptado do trabalho de Boronat e outros [BRC06]. Uma companhia possui ônibus (classe `Bus`), os quais são utilizados

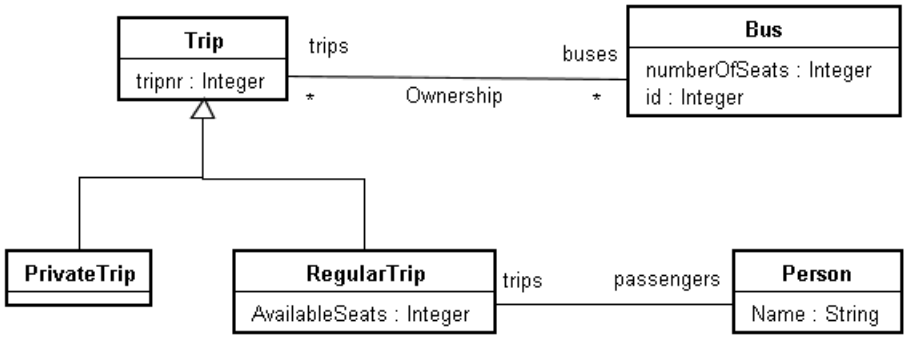

Figura 2.3: Diagrama-exemplo do uso de OCL: proibição de overbooking

para a realização de excursões (classe `Trip`), que podem ser convencionais (classe `RegularTrip`) ou particulares (classe `PrivateTrip`). A realização de uma excursão pode necessitar de mais de um ônibus, onde cada um possui um número específico de lugares. Em excursões convencionais, as passagens são compradas individualmente. Em excursões particulares, todo o ônibus é alugado. Como representar neste diagrama que, nas excursões convencionais, é proibida a venda de quantidade de passagens superior à disponibilidade de lugares nos ônibus alocados para a excursão (i.e., overbooking)? Uma possível solução em OCL:

```
context Bus inv overbooking:
self.trips->select(t|t.oclIsType(RegularTrip))->forAll(t|
   t.passengers->size() <= t.buses->collect(b | b.numberOfSeats)->sum() )
```

Traduzindo: Dentre as excursões que cada ônibus esteja escalado para fazer (`self.trips`), selecionar todas as excursões convencionais (operação `select(t|t.oclIsType(RegularTrip))`). Dessa coleção de excursões, é estabelecido que, para todas elas (operador `forAll`), o número de passageiros deve ser menor ou igual à soma do número de assentos de todos os ônibus que integrem a excursão (`t.buses->collect(b | b.numberOfSeats)->sum()`).

Discussão: Linguagens restritivas e construtivas. Segundo Baar [Baa05], OCL é uma linguagem de especificação *restritiva*. Em tais linguagens, uma pós-condição restringe o conjunto de estados possíveis de um software terminada a execução da operação respectiva. A intenção não é descrever como esse *pós-estado* é construído a partir de um estado anterior à sua execução, ou *pré-estado*. Nesse aspecto, OCL é diferente de linguagens formais de especificação *construtivas*, como a linguagem *B*. Nessa linguagem, a formulação da pós-condição é feita utilizando um pseudo-código que especificará o comportamento da operação na forma de um algoritmo, explicitando-se cada passo na transição do pré-estado para o pós-estado. Esse pseudo-código assemelha-se a linguagens de programação imperativas e suas estruturas básicas de controle.

Quanto à verificação de restrições em uma implementação Java, para pré e pós-condições é trivial: devem ser verificadas antes e depois da execução dos métodos, respectivamente. Em relação aos invariantes, Meyer afirma que devem ser verdadeiros após a criação de uma instância, antes e depois de qualquer chamada remota a um método da classe. Briand e outros [BDL04] utilizaram essa definição, verificando-os antes e depois de todas as operações públicas de uma classe.

Com relação à herança de invariantes, foi constatado um problema. O *Princípio de Liskov* afirma que as pré-condições podem ser enfraquecidas, enquanto que as pós-condições podem ser fortalecidas. Segundo o paradigma de Projeto por Contrato [Mey97], a avaliação de pré e pós-condições em classes que possuem invariantes deve ser feita com a conjunção desses invariantes:

- pré-condição: `invariante E pré-condição`;
- pós-condição: `invariante E pós-condição`.

Warmer e Kleppe [WK03], em seu manual sobre OCL e MDA, afirmam que "Um invariante de uma super-classe é herdado por suas sub-classes. Uma sub-classe pode fortalecer o invariante, mas não pode enfraquecê-lo.". O mesmo é afirmado por Meyer [Mey97], conforme o qual "Os invariantes das classes ancestrais são adicionados aos da própria classe. (...) os invariantes de todos os ancestrais se aplicam, direta ou indiretamente (à classe).". Mas, se for permitido o fortalecimento de invariantes, o Princípio de Liskov será violado: caso o invariante seja fortalecido, não há problema em fortalecer a pós-condição; mas esse fortalecimento do invariante fará com que a pré-condição também seja fortalecida, violando o Princípio. Para respeitá-lo, é necessário que os invariantes de uma super-classe sejam preservados em suas sub-classes (i.e., a implicação de invariantes deve ocorrer nos dois sentidos: da super-classe para a sub-classe, e vice-versa).

Neste trabalho, o foco da discussão sobre OCL restringiu-se a sua versão 2.0, a menos que a versão seja explicitamente mencionada no texto. Para maiores informações sobre OCL, vide sua especificação [OCL03] e o livro de Warmer e Kleppe [WK03].

Capítulo 3

Uso de OCL na Engenharia Direta

O processo de engenharia direta envolve a geração de um ou mais artefatos de software que sejam mais parecidos com o programa final a ser implantado (por exemplo, código-fonte), na forma e em nível de detalhe, do que os artefatos utilizados como insumo desse processo (por exemplo, diagramas de classe de um modelo UML) [SK04]. Mais especificamente, a geração de código é a tarefa de criação do código-fonte, em uma linguagem de programação, a partir de modelos existentes. Nos últimos anos e de forma crescente, algumas organizações têm usado modelos UML para impulsionar a geração de código e outros artefatos de software. Geralmente, isso é feito a partir de diagramas de classe, que são usados para gerar interfaces entre os vários componentes de uma aplicação distribuída [HS05]. Essa geração automática de código a partir de um modelo simplifica a implementação, agilizando o desenvolvimento de software. Isso ainda é mais facilitado pelo fato de muitos elementos do meta-modelo UML possuírem contra-partidas em Java, tais como: classes, atributos e operações. Entretanto, esse não é o caso para alguns elementos da UML, como associações[1] e restrições.

Ahrendt e outros [ABB+05] afirmam que a OCL é uma notação apropriada para combinar projetos orientados por objetos e uma maior precisão em especificações, anexando-se restrições a diagramas de classe. Rumbaugh e outros declaram que, quando um modelo contém restrições, ele não necessariamente informa o que deve ser feito caso essas restrições sejam violadas. Sendo um modelo prescritivo, ele declara o que deveria acontecer, ficando a cargo do implementador a tarefa do que fazer em caso de violações. De qualquer forma, se um modelo puder auxiliar a produzir um programa que é correto por construção ou puder ser verificado como correto, então o modelo serviu a seu propósito [RJB05]. Henderson-Sellers [HS05] afirma que o uso de automação é a forma mais eficaz de se atingir esse resultado, aumentando a produtividade dos desenvolvedores. Isso é conseguido por meio da mecanização de tarefas repetitivas, como a geração automática de código a partir de modelos. Ainda segundo ele, "há muitos exemplos de sistemas da indústria baseados em modelos que realizam a geração completa e automática de código, então isso não é meramente uma promessa para o futuro, e sim, estado-da-arte".

[1] Dificuldades para a recuperação desses relacionamentos são apresentadas na Seção 4.13, problema 4.

3.1 Estado-da-arte do uso de OCL

Foram estudadas diferentes ferramentas que utilizam anotações OCL, presentes em diagramas de classe UML, para realizar algum processamento que auxilie a engenharia direta. A escolha por Java é de ordem prática: maior experiência com a linguagem e farta documentação disponível, além de ser uma linguagem moderna e poderosa, e que oferece muitos recursos. Além disso, Ahrendt e outros [ABB+05] afirmam que "Java é adequada para interação com outras ferramentas, escritas em Java ou não".

As abordagens de engenharia direta analisadas foram estudadas segundo a ótica da disponibilidade de seus recursos e a facilidade de sua adaptação a novas necessidades. Dessa forma, os critérios por meio dos quais foram analisadas refletem essa visão.

Quando mencionadas, as operações OCL serão apresentadas em um dos seguintes formatos:

- `<operação>(parâmetros):<valor de retorno>`
- `<classe>::<operação>(parâmetros):<valor de retorno>`
- `<pacote>::...::<classe>::<operação>(parâmetros):<valor de retorno>`

Em caso de dúvidas sobre OCL, sugerimos a consulta à Seção 2.4 dessa dissertação, à especificação da linguagem [OCL03] ou ao livro de Warmer e Kleppe [WK03].

3.2 USE

USE (UML-based Specification Environment) [GBR03, RG03, BGG04, GBR05] é uma ferramenta de verificação de inconsistências em diagramas de classe UML, desenvolvida na Universidade de Bremen, Alemanha.

A USE não gera código a partir do diagrama e suas restrições, mas é capaz de interpretar expressões OCL, utilizando-a como uma linguagem de consulta sobre um dado diagrama de classe UML, exibindo o resultado das expressões fornecidas. Essencialmente, o propósito dessa ferramenta é a verificação de modelos. Isso é feito por meio da animação, realização de testes e a validação de diagramas de classe UML e suas respectivas restrições OCL, focando-se nos estágios iniciais do processo de desenvolvimento. Dessa forma, podem ser identificados, antecipadamente, defeitos presentes em diagramas de classe (por exemplo, se uma classe não pode possuir instâncias). Entretanto, como será visto adiante, USE também pode ser usada para verificar se uma dada implementação em Java não viola as restrições presentes no diagrama de classe do modelo respectivo.

USE possibilita a validação de diagramas de classe UML anotados com OCL. Em seu trabalho, Bauerdick e outros [BGG04] utilizaram essa ferramenta para identificar, na super-estrutura da especificação da UML 2.0 [UML05], erros de sintaxe e de verificação de tipos em algumas de suas regras de boa formação e definições de operações. Por exemplo, seja a seguinte definição da operação `Kernel::Package::visibleMembers()` (adaptada por Bauerdick e outros, da versão presente à pág. 103 da especificação citada):

```
visibleMembers():Set(Kernel_PackageableElement)=
    member->select( m | self.makesVisible(m))
```

Esta operação deve retornar quais membros de um pacote podem ser acessados fora dele. Há um problema com a definição dessa operação. Seja o diagrama de classe presente na Figura 3.1, extraída do trabalho dos autores. O identificador `member` é o nome do papel que a super-classe `NamedElement`

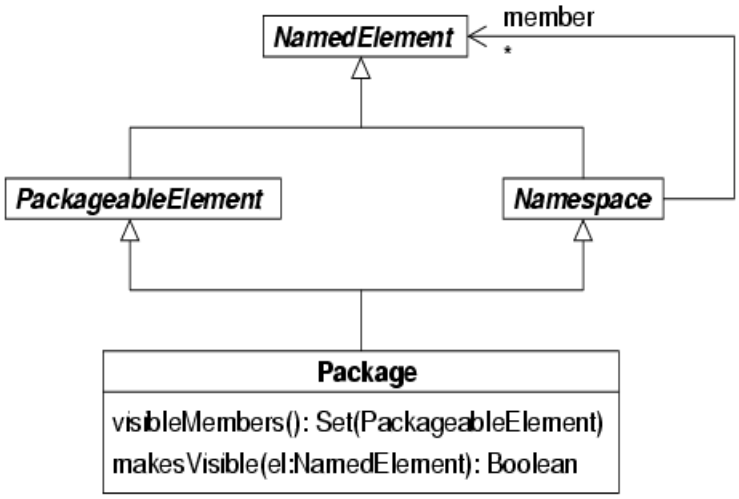

Figura 3.1: Diagrama de classe para contexto da operação `visibleMembers()`

assume em sua associação com uma de suas sub-classes, `Namespace`. A operação `select` retornará um conjunto de elementos do tipo `NamedElement` que atendam à restrição `self.makesVisible(m)`. Entretanto, segundo o diagrama da Figura 3.1, a operação `visibleMembers()` deveria produzir como resultado um conjunto de elementos do tipo `PackageableElement`, sub-tipo de `NamedElement`, evidenciando a inconsistência entre a regra de boa-formação do modelo e um de seus diagramas de classe.

Em outra utilização da ferramenta, Richters e Gogolla [RG03] fizeram a validação de uma implementação de software, em relação às restrições especificadas no modelo UML respectivo. Utilizando um modelo e suas restrições, a ferramenta gera um *aspecto* [Lad03]. Quando compilado com a implementação Java do modelo respectivo e executado, esse aspecto é responsável por enviar à USE mudanças de estado ocorridas durante a execução do programa, para que a ferramenta faça a verificação das restrições. Em caso de violação de alguma restrição, a USE notifica o usuário, apontando a restrição violada. Invariantes são verificados antes e depois da execução de cada método público Java, bem como depois da invocação de construtores. Pré-condições são verificadas antes da chamada dos métodos respectivos. Pós-condições, após a chamada.

Um inconveniente do uso da ferramenta é a necessidade que as informações de entrada – diagrama de classe e suas restrições – estejam num formato textual próprio da USE, não suportando a leitura de um modelo já construído em outro formato, como XMI, por exemplo.

3.3 DOT

Na Universidade de Tecnologia de Dresden, Alemanha, foi desenvolvida a ferramenta *DOT* (Dresden OCL Toolkit) [HDF00, LO03]. Ela é um compilador capaz de realizar a análise sintática (parsing)

de expressões OCL, verificação de tipos e avaliação de restrições presentes em modelos comuns, recursos também suportados pela USE (mostrada anteriormente). Entretanto, a DOT também é capaz de gerar, automaticamente, o código Java a partir das expressões OCL e do modelo UML fornecidos. É baseada nas especificações da UML 1.4 e da OCL 2.0.

A DOT realiza a geração de código e instrumenta o código-fonte da aplicação, para que sejam verificadas as restrições em tempo de execução. Isso ocorre em quatro etapas: 1) a partir de um arquivo XMI ou de comentários no código-fonte, realiza-se a análise sintática, resultando em uma árvore sintática abstrata (ou AST); 2) a AST das expressões, em OCL, é transformada, convertendo-se algumas das operações OCL para um sub-conjunto de operações dessa linguagem, a fim de simplificar a posterior geração de código (por exemplo, `select` é transformado em `iterate`); 3) navegando pela AST, o gerador de código produz as expressões correspondentes em Java; 4) essas expressões são então inseridas na aplicação, para que sejam testadas em tempo de execução.

A inserção do código para verificação das expressões na aplicação é feita conforme cada tipo de restrição – invariantes, pré-condições ou pós-condições. Para os invariantes, em vez de serem verificados antes e depois da execução de cada método público e após cada construtor, sua execução ocorre sempre no fim de métodos que modifiquem o valor de campos utilizados pelos invariantes. Para cada campo presente no código, são adicionados *observadores* (mecanismo de desenho Observer [GHJV94]) que são notificados se o valor do campo respectivo mudar. Isso determina quais invariantes devem ser verificados devido às mudanças ocorridas, disparando por sua vez a execução dos invariantes relacionados.

Para a execução das pré-condições e pós-condições, os métodos são renomeados e embrulhados (wrapped). Além disso, nos métodos embrulhadores (wrapper), são criados três *fragmentos*, nos quais as informações necessárias à execução dessas restrições são distribuídas. O estilo (template) utilizado pela ferramenta para a inserção de pré e pós-condições segue abaixo (adaptado do trabalho de Briand e outros [BDL04]):

```
public class ClasseA {
    // Método original, embrulhado.
    public tipoDeRetorno metodoX_original() {
        ...
    }
    // Método embrulhador.
    public tipoDeRetorno metodoX() {
        // TRANSFER FRAGMENT
        ...
        {
            // PRE FRAGMENT
            ...
        }
        // Invocação do método original.
            result = metodoX_original();
        {
```

```
            // POST FRAGMENT
            ...
        }
        return result;
    }
}
```

A função de cada fragmento é explicada abaixo:

TRANSFER FRAGMENT: Nesse fragmento, antes da execução do método original, são salvos os valores dos campos que se encontrem sufixados pelo operador `@pre`. Tipicamente, serão utilizados para comparação com seu valor do campo após a execução do método (vide exemplo de especificação de uma pilha, operação `desempilhar()`, na Seção 2.4).

PRE FRAGMENT: Armazena o código para verificação de pré-condições, imediatamente antes da execução do método original.

POST FRAGMENT: Armazena o código para verificação de pós-condições, imediatamente após a execução do método original. Se for o caso, pode utilizar os valores anteriores dos campos, armazenados no primeiro fragmento.

Verheecke [Ver01] aponta que, com essa abordagem, não é possível renomear construtores existentes. Pior ainda, uma classe pode não possuir nenhum construtor, fazendo com que o compilador Java crie um construtor padrão, que também não pode ser embrulhado.

Outro recurso da DOT é o uso de mecanismos de reflexão Java, em tempo de execução da aplicação sendo instrumentada, para determinar detalhes de sua implementação (por exemplo, a forma que uma coleção presente numa expressão OCL é implementada na aplicação). Reflexão é usada para identificar os tipos de retorno de métodos, os tipos dos argumentos desses métodos e os tipos dos campos das classes. Para ilustrar essa necessidade, seja a seguinte restrição em OCL:

```
context X inv:
  self.obtemDados(9)->count() <= 1
```

Para gerar a versão Java dessa restrição, é necessário obter duas informações. A primeira, no âmbito de OCL, é o tipo da coleção retornada pela operação (a simples investigação do modelo pode responder a isso). A segunda informação, obtida por reflexão, é qual o tipo do argumento na execução de `obtemDados()`. Se o argumento do método executado em Java for um inteiro, nada precisa ser feito. Entretanto, caso seja do tipo `java.lang.Integer`, o código em Java para verificação desse invariante deve converter o argumento `9` para `new Integer(9)`. A recuperação desses tipos a partir do diagrama de classe do modelo UML respectivo não foi considerada pelos autores. Segundo eles, os tipos nesses diagramas podem não estar especificados em nível detalhado o suficiente para serem fiéis à implementação respectiva. Por fim, a opção pelo uso de reflexão em vez de análise estática do código-fonte se deveu à simplicidade da primeira abordagem.

A DOT tem sido utilizada na arquitetura de diferentes ferramentas (por exemplo, protótipo da abordagem de Verheecke, KeY Tool), nas quais opera como um "intermediário" entre as funcionalidades de modelagem e as de inserção de código dessas ferramentas.

3.4 Abordagem de Verheecke

Outra ferramenta foi desenvolvida por Verheecke [Ver01, VS02], na Universidade de Vrije, Bélgica. Ela analisa o modelo e as restrições presentes, gerando o código Java correspondente. As restrições são implementadas como classes explícitas. Além disso, usa internamente a DOT (cuja versão utilizada pelo autor suportava a versão 1.3 da OCL) para converter as expressões de OCL para Java, encontrando-se integrada à ferramenta de código-aberto *Argo/UML* [arg05].

Segundo o autor, o desenvolvedor deveria criar a aplicação verificando suas próprias restrições, usando a ferramenta para dar suporte ao processo de desenvolvimento durante as atividades de desenho. Entretanto, como essas restrições não especificam como devem ser implementadas, a ferramenta exige a interação desse desenvolvedor, aproveitando seu conhecimento e experiência com o domínio do problema sendo modelado.

Cada restrição OCL presente no modelo sendo analisado é mapeada para uma classe Java. Para sua correta verificação, são identificadas dez características em cada restrição. O desenvolvedor pode alterá-las, validando ou recusando as inferências realizadas pela ferramenta:

1. Nome: nome da restrição; utilizado também como nome da classe de restrição a ser gerada.
2. Expressão: restrição OCL propriamente dita; a partir dela são inferidas pela ferramenta as demais características.
3. Classe de contexto: classe do modelo onde a restrição foi definida; classe a partir da qual se faz a navegação da restrição.
4. Tipo de restrição: *invariante*, *pré-condição* ou *pós-condição*.
5. Ação: ação que deve ser executada ao se detectar uma violação. Ações possíveis: *ignorar* (nada é feito), *avisar* (após a violação, o usuário da aplicação gerada é notificado com uma mensagem), *proibir* (antes que a mudança de valor viole a restrição, ela é verificada, preservando o estado do objeto), *reparar* (após a violação, o estado anterior do objeto é recuperado), ou uma ação específica definida pelo usuário.
6. Prioridade: prioridade de verificação da restrição; útil nos casos que o programa viola mais de uma restrição, pois possibilita ao usuário da ferramenta especificar a ordem de execução de ações (mostradas no item anterior).
7. Pontos de inserção: conjunto de métodos nos quais as restrições são verificadas.
8. Pontos de verificação: indica onde verificar a restrição nos pontos de inserção: *antes* da execução do método respectivo (no caso de uma pré-condição), ou *depois* (no caso de uma pós-condição).
9. Forma traduzida: expressão booleana em Java correspondente à restrição OCL, gerado pela ferramenta.
10. Chave: habilita e desabilita a verificação de uma restrição num dado ponto de inserção. Isso é útil nos casos em que um dado método implementado viole interna e temporariamente a restrição respectiva, retornando a um estado consistente terminada sua execução. Por exemplo, quando uma restrição estipule que dois campos devem sempre possuir o mesmo valor, a atribuição a cada um dos campos ocorre uma de cada vez. Nesse caso, desabilita-se a restrição, atribui-se o primeiro valor e habilita-se a restrição novamente para que ela seja verificada após o segundo valor ser atribuído.

Para auxiliar o levantamento das características citadas, as restrições identificadas são classificadas segundo seis categorias. Os exemplos mostrados são baseados no diagrama representado na Figura 3.2, extraído da dissertação do autor [Ver01]:

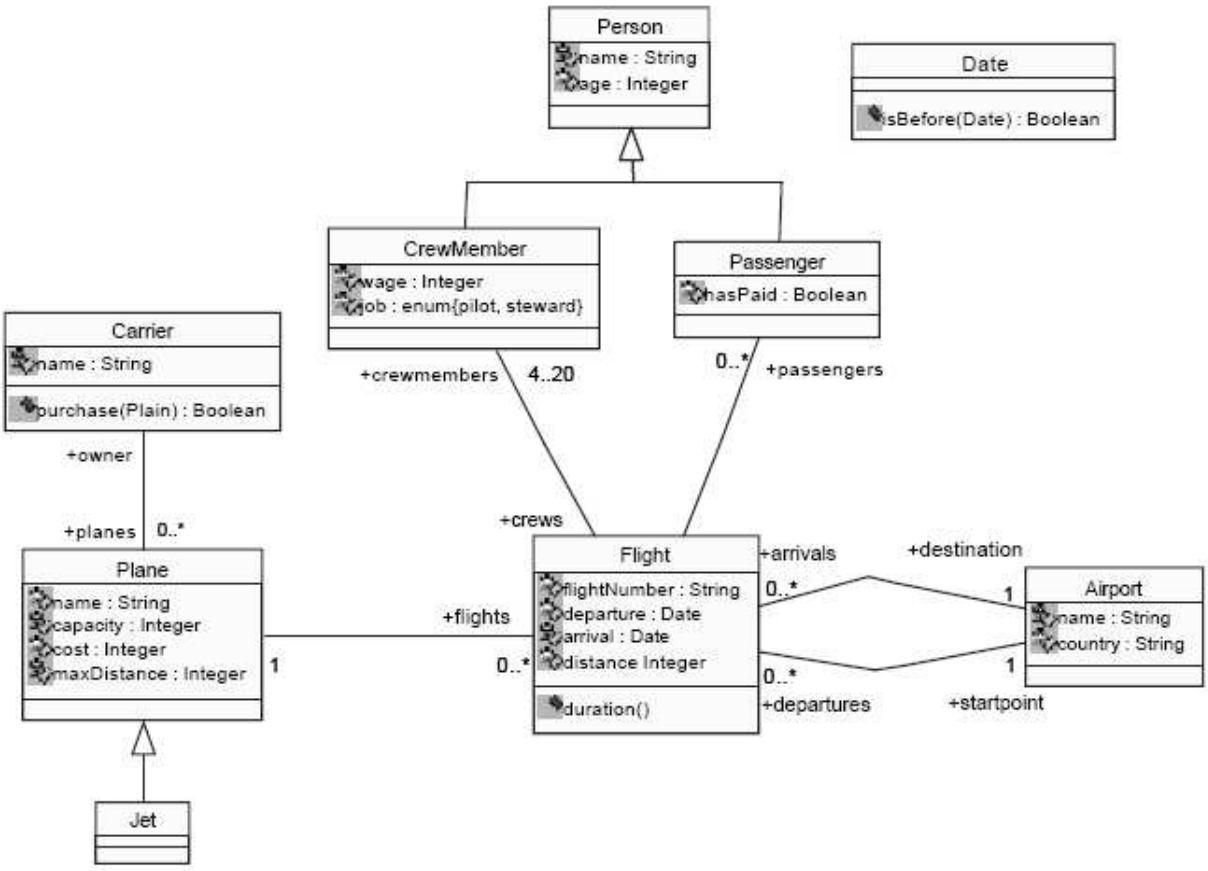

Figura 3.2: Diagrama de classe utilizado nos exemplos de categorias.

1. Invariantes em atributos, extremidades de associação, operações ou métodos de consulta *de uma única classe*. É feita a monitoração de construtores e de todos os métodos que alterem as propriedades envolvidas (por exemplo, atributos), implicando que todos os pontos de inserção se localizem na classe de contexto. Exemplo: "Uma pessoa deve ter, pelo menos, 18 anos de idade".

   ```
   context Person inv ageConstraint: age >= 18
   ```

   Abaixo, segue o resultado da conversão do exemplo para sua versão em Java:

   ```
   public class AgeConstraint extends Constraint {
       public static void assert (Person person)
           throws ConstraintViolation {
               int age = person.getAge();
               if (age < 18)
                   throw new ConstraintViolation ("A
                   violation of ageConstraint occurred")
   ```

```
        }
}
```

2. Invariantes em associações *entre duas classes*. São restrições aplicadas a associações entre duas classes (por exemplo, multiplicidade e bidirecionalidade de associações). São monitorados os construtores e métodos que alterem a associação entre dois objetos das classes respectivas. Isso implica que os pontos de inserção se localizam nas classes associadas.
3. Invariantes em atributos, extremidades de associação, operações ou métodos de consulta de classes associadas. As restrições envolvem elementos de classes que podem estar espalhadas pelo diagrama de classe. Seus pontos de inserção são os métodos que alterem as propriedades envolvidas (por exemplo, atributos), construtores da classe de contexto e todos os métodos que modifiquem as associações entre objetos das classes envolvidas. Exemplo: "A distância percorrida em um vôo deve ser menor que a distância máxima que um avião pode voar sem precisar reabastecer."

   ```
   context Flight inv: self.distance < self.plane.maxDistance
   ```
4. Invariantes em atributos, extremidades de associação, operações ou métodos de consulta *de coleções associadas*. São restrições que se referem ao tamanho de coleções. Seus pontos de inserção são todos os métodos que alterem o número de elementos das coleções associadas. Exemplo: "O número de passageiros em um vôo deve ser de, no máximo, a capacidade do avião que é alocado para aquele vôo."

   ```
   context Flight inv: self.passengers->size() <= self.plane.capacity
   ```
5. Invariantes em atributos, extremidades de associação, operações ou métodos de consulta *de elementos internos às coleções*. São restrições que se referem aos elementos que estão dentro das coleções, sendo necessário avaliar a restrição iterativamente para todos os elementos. Exemplo: "Todos os passageiros de um vôo devem ter pago a passagem."

   ```
   context Flight inv: self.passengers->forAll(p:Passenger | p.hasPaid)
   ```
6. Pré e pós-condições. A verificação dessas restrições é trivial: os pontos de inserção são os próprios métodos respectivos. Exemplo: "A duração de um vôo deve ser sempre superior a 0."

   ```
   context Flight::duration(void)
     post: result > 0
   ```

3.5 SPACES

SPACES (SPecification bAsed Component tESter) é a ferramenta desenvolvida por Barbosa e outros [BAMF04], na Universidade Federal de Campina Grande. Ela faz uso de especificações em UML (diagramas de classe, de casos de uso e de seqüência), anotadas com OCL, para derivar *casos de teste* automaticamente.

O modelo UML é carregado na ferramenta utilizando seu analisador sintático (parser) XMI, construído em conformidade com a versão 1.2 desse formato. As restrições presentes no diagrama de classe são utilizadas na geração de *oráculos* para cada caso de teste. Esses oráculos são responsáveis pela determinação do sucesso ou falha em um caso de teste. As pré-condições determinam as

condições necessárias para a execução de cada cenário de uso de uma funcionalidade a ser testada; as pós-condições verificam se as mudanças de estado previstas ocorreram de fato[2]. Caso essas restrições não estejam presentes no modelo, a ferramenta requer a interação do usuário para que as escreva em OCL. Abaixo, segue um exemplo da conversão de um invariante em OCL para sua versão correspondente em Java, realizada pela ferramenta: "A idade mínima para todos os empregados de uma empresa é de 18 anos." Em OCL:

```
context Empresa inv idadeEmpregados:
  self.empregados->forAll(p:Pessoa | p.idade >= 18)
```

Em Java:

```
boolean verifica_idadeEmpregados(Empresa emp){
  boolean result = true;
  java.util.Collection empregados = emp.getEmpregados();
  java.util.Iterator empregadosIt = empregados.Iterator();
  Pessoa p;
  while( empregadosIt.hasNext() ){
    p = (Pessoa)empregadosIt.next();
    if(! (p.getIdade() >= 18) ){
      result = false;
      break;
    }
  }
  return result;
}
```

Como resultado final de geração, a SPACES produz um *componente de teste*, que é um programa que contém as classes de teste geradas para as diversas funcionalidades a serem testadas no software respectivo.

3.6 ocl2j

Outra ferramenta, a *ocl2j*, foi desenvolvida por Briand e outros [BDL04, DBL05], na Universidade de Carleton, Canadá. Ela visa converter contratos OCL, presentes em diagramas de classe de um modelo UML, adicionando-os a programas Java. Para isso, ela utiliza *AspectJ*, uma implementação Java de *POA* (Programação Orientada por Aspectos) [Lad03]. Dessa forma, ela faz a verificação automatizada de contratos OCL durante a execução de uma aplicação Java a ser testada. Suporta a versão 1.4 da OCL, bem como Java 1.4.

Em linhas gerais, ocl2j opera da seguinte maneira: as informações necessárias à ferramenta são recuperadas do diagrama de classe UML e do código-fonte da aplicação Java a ser instrumentada

[2] A função dos invariantes na ferramenta não é mencionada pelos autores. Presume-se que sejam verificados ao fim da execução de cada caso de teste, reforçando as pós-condições respectivas, ou que de fato não sejam verificados.

(i.e., que receberá os contratos OCL convertidos, para verificação em tempo de execução); todas as expressões OCL encontradas são analisadas, gerando ASTs que serão usadas para criar o código correspondente em Java; a aplicação-alvo é então instrumentada com o código gerado dos contratos, utilizando aspectos feitos em AspectJ.

Para realizar a conversão de OCL para Java, sempre que possível utilizam-se métodos Java que se assemelhem às respectivas operações em OCL. Nesse sentido, a operação `size()` da interface `java.util.Collection` mapeia diretamente a operação OCL `Collection::size():Integer`. Quando esse mapeamento não existe, a funcionalidade que falta é implementada nos aspectos. Por exemplo, a operação `Collection::count(object:T):Integer`, que "conta o número de vezes que o objeto ocorre na coleção respectiva". Para implementá-la, uma classe interna ao código do aspecto contém um método estático cuja assinatura possui dois parâmetros: a coleção que será varrida e o objeto a ser contado.

Para a conversão dos tipos dos atributos presentes nas expressões OCL, são analisados: os tipos presentes na expressão OCL; os tipos dos elementos da coleção presentes na aplicação Java, bem como as características da expressão. É feita uma avaliação dos tipos presentes no código-fonte Java também. Por exemplo, a expressão OCL `someCollection->includes(5)`, ao ser convertida para Java, considerará o argumento `5` como o tipo embrulhador Java `Integer` em vez do tipo primitivo `int`. Isso acontece porque elementos de coleções Java devem ser referências. Os tipos primitivos da OCL são convertidos para os tipos primitivos Java semelhantes, apenas quando seus valores são usados em operações lógicas, de adição, de multiplicação e unárias.

Quanto à conversão de coleções, o tipo OCL `Collection` é mapeado diretamente para a interface de coleção `java.util.Collection`. Os tipos `Bag` e `Sequence` são convertidos para a interface `java.util.List`, enquanto que o tipo OCL `Set` é mapeado diretamente para a interface `java.util.Set`[3]. Em relação à existência de métodos em Java similares às operações de cada tipo de coleção OCL, os autores observaram três situações possíveis:

- Existência de mapeamento direto. Exemplo: `Collection::isEmpty():Boolean` (em OCL); `boolean java.util.Collection.isEmpty()` (em Java).
- Não há mapeamento direto, mas sua funcionalidade pode ser derivada de outras operações existentes. Exemplo: `Sequence::prepend(o:Object):Sequence` (em OCL); `void add(int index, Object o)`, com `index` igual a zero (em Java).
- Operações OCL que iteram pelos elementos das coleções. Não havendo uma contra-parte em Java, sua funcionalidade deve ser instrumentada no código do aspecto. Do tipo OCL `Collection`, tem-se: `exists`, `forAll`, `isUnique`, `collect` e `iterate`. Dos tipos `Set`, `Bag` e `Sequence`, tem-se: `select`, `reject` e `sortedBy`. Exemplo: em OCL, a operação
  `Set::select(ex:OclExpression):Set`
  é convertida para Java

```
Set result = new HashSet();
private Set select(Collection c) {
    for(Iterator i = c.iterator(); c.hasNext();){
      ElementType e = (ElementType)i.next();
```

[3]O tipo `OrderedSet` não é mencionado, pois não existia na versão da OCL suportada pela ferramenta.

```
        if (OclExprInJava) {
          result.add(e);
        }
      }
    return result;
  }
```

Na linguagem OCL, o uso da palavra-chave `let` permite a definição de uma constante que pode ser empregada dentro de outra expressão. Segundo os autores, esse recurso não é suportado pela ferramenta.

Em seu trabalho, Verheecke [Ver01] aponta algumas desvantagens do uso de POA para instrumentar as restrições OCL, como o que foi realizado na ocl2j:

- com o acréscimo de novos comportamentos e funcionalidades a uma aplicação, utilizando-se aspectos, o desenvolvedor perde o controle sobre o código, pois essas mudanças ocorrem à sua revelia ("caixa-preta");
- a gestão de configurações do projeto deve gerir duas árvores de código-fonte;
- o desenvolvedor deve estar atento para editar apenas a versão original do código-fonte, não instrumentado;
- é necessário que se execute a ferramenta de instrumentação após cada mudança do código-fonte, e não apenas quando as restrições mudarem;
- conflitos entre o código-fonte original e o código-fonte instrumentado são difíceis de se localizar e resolver;
- traços dinâmicos de exceções (stack traces of runtime exceptions) apontam para o código-fonte modificado, ficando a cargo do desenvolvedor localizar, no código original, o trecho de código correspondente;
- a maioria dos desenvolvedores Java provavelmente têm uma profunda aversão de usar um ambiente Java dedicado apenas para verificar restrições OCL, exigindo-se, ainda, o conhecimento de conceitos de POA.

Acrescenta-se às desvantagens acima a perda do raciocínio modular. Segundo Wand [Wan03], linguagens de POA violam raciocínio modular, pois seu uso impede a dedução do comportamento de um módulo composto a partir de suas partes constituintes. Os aspectos podem mudar radicalmente o funcionamento do módulo, tornando-o dependente do contexto onde é aplicado. Para resolver estes problemas, Verheecke [VS02] propõe armazenar as restrições OCL, já convertidas para Java, em classes separadas, onde cada classe representa uma restrição.

3.7 KeY Tool

KeY Tool [ABB+05, GH04, GHL04, GL05, HJR02] é uma ferramenta de especificação formal e verificação de programas no desenvolvimento de software UML. Foi desenvolvida na Universidade de Tecnologia de Chalmers, Suécia, como um plug-in para uma ferramenta *CASE* (Computer Aided Software Engineering) comercial[4].

[4] *Borland Together Control Center*. Para maiores informações, vide http://www.borland.com/together/index.html

Uma das metas do projeto KeY é a popularização de métodos formais na indústria de desenvolvimento de software, por meio da utilização de restrições OCL, focando sua aplicação em diagramas de classe de um modelo UML. A KeY Tool suporta um sub-conjunto da linguagem Java, chamado *Java Card*, versão 2.2. Ele exclui certas características (por exemplo, clonagem e carregamento dinâmico de classes), com uma *API* (Application Programming Interface) reduzida, sendo voltado para execução em *smart cards* e outros dispositivos com memória limitada, tais como sistemas embutidos.

Os autores advogam a utilidade da ferramenta em três cenários de aplicação:

1. Modelagem precisa. Se por um lado, alterações na implementação ocorrem freqüentemente, por outro lado, modelos (e seus diagramas de classe anotados com restrições OCL) são menos propensos a mudar. O uso de restrições OCL em diagramas de classe de um modelo de desenho, removendo suas ambigüidades, pode resultar em ganhos significativos na identificação e remoção de defeitos durante o processo de desenvolvimento. Nesse sentido, a KeY Tool fornece gabaritos (*KeY Idioms* e *KeY Patterns*) para auxiliar usuários menos experientes com a linguagem. Além disso, é disponibilizado o recurso da tradução de restrições em OCL para descrições em linguagem natural, em inglês [HJR02]. Para exemplificar, o seguinte texto (referente ao diagrama de classe da Figura 3.3)

   > "The following invariant holds for all `BasicCreditCards`: the `bankLine` of the `BasicCreditCard` is greater than or equal to zero."

   é obtido da seguinte expressão em OCL:

   ```
   context BasicCreditCard inv: self.bankLine >= 0
   ```
2. Desenvolvimento de softwares críticos. A extensão dos danos (humanos e/ou materiais) que podem ser causados no caso de implementações defeituosas justificam o investimento de esforço adicional em verificação formal.
3. Ferramenta de ensino. A ferramenta pode ser usada como um recurso auxiliar no ensino de provas de teoremas, utilizando lógica de predicados. Além disso, pode auxiliar no aprendizado da própria OCL.

A KeY Tool faz a verificação de restrições presentes num diagrama de classe de um dado modelo, partindo da premissa que as relações entre classes implicam relações entre as restrições correspondentes. Dessa forma, as restrições podem ser analisadas independentemente de como forem implementadas. Além disso, a ferramenta faz a verificação de consistência entre implementações Java e restrições OCL anotadas no diagrama de classe, utilizando um provador de teoremas. Para tanto, o diagrama de classe, as restrições anotadas, bem como a implementação Java são utilizadas na construção automatizada de provas formais de sua consistência. Para a conversão das expressões OCL para Java, utilizam-se as funcionalidades da ferramenta DOT (apresentada na Seção 3.3).

Giese e outros [GHL04, GL05] adicionaram à KeY Tool a funcionalidade de simplificação de restrições OCL através do uso de algumas regras pré-definidas. Por exemplo, sejam duas classes, `PieChart` e `BarChart`, e a seguinte expressão num contexto qualquer:

```
Set{PieChart,BarChart}->
  forAll( s | s.allInstances()->forAll(i | i.statistics->size() <= 1) )
```

Essa expressão verifica se todas as instâncias das classes mencionadas possuem, no máximo, uma estatística (atributo `statistics`). Uma característica que deva ser obedecida por todos os objetos de uma classe pode ser escrita como um invariante. A ferramenta transforma a expressão em:

```
context PieChart inv: self.statistics->size() <= 1
context BarChar inv: self.statistics->size() <= 1
```

As expressões OCL dos exemplos que se seguem baseiam-se no diagrama representado na Figura 3.3, extraído do trabalho dos autores [ABB+05]. Esse diagrama modela cartões de crédito (`BasicCreditCard`), na qual há cartões com limite de crédito mais restrito (`JuniorCard`), bem como cartões cuja utilização oferece bônus (`BonusCard`). Essa utilização é representada pela execução da operação `debit`.

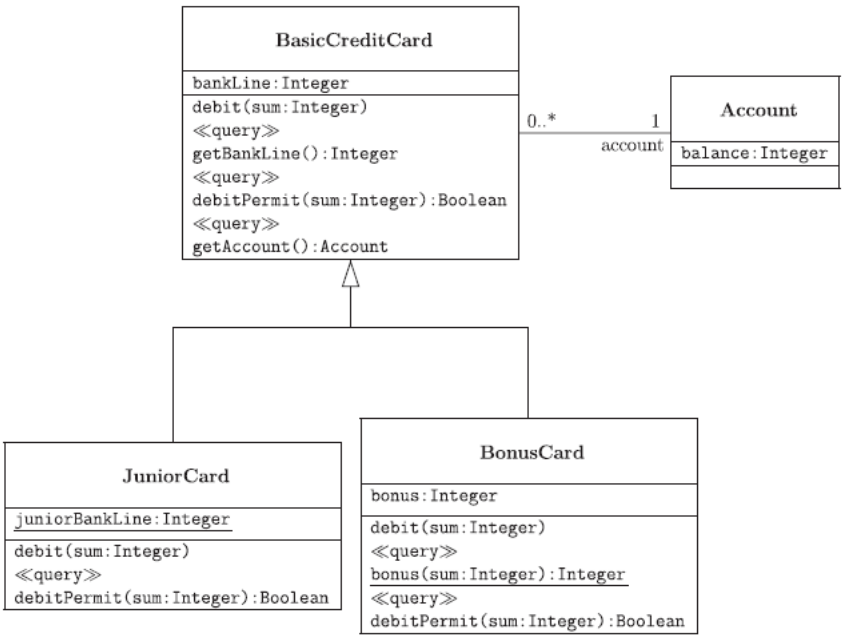

Figura 3.3: Diagrama de classe utilizado nos exemplos de uso da KeY Tool.

A KeY Tool fornece mecanismos de desenho que podem ser instanciados pelo desenvolvedor, contendo restrições OCL geradas automaticamente. Um exemplo simples dessas restrições é mostrado a seguir. Dada a implementação da operação `getBankLine()`:

```
public int getBankLine() {
  return component.getBankLine();
}
```

gerada pela ferramenta CASE da qual faz parte, a KeY Tool gera a seguinte pós-condição para garantir a consistência entre especificação e implementação:

```
context BasicCreditCard::getBankLine():Integer
  post: result = self.component.getBankLine()
```

Para a análise de dependências entre os invariantes, pré e pós-condições, a KeY Tool suporta *sub-tipagem estrutural* (structural sub-typing) e *sub-tipagem comportamental* (behavioural sub-typing). Na sub-tipagem estrutural, é verificado se a conjunção de todos os invariantes de uma sub-classe implica todos os invariantes de sua super-classe[5]. Por exemplo, os invariantes de `JuniorCard` – "o limite de um dado cartão de crédito júnior deve ser de, no máximo, o limite estabelecido para esse tipo de cartão; a conta de um cliente pode ser devedora de, no máximo, o limite estabelecido para esse tipo de cartão; o limite desse cartão deve ser de, no mínimo, zero":

```
context JuniorCard
  inv: JuniorCard.juniorBankLine >= self.bankLine
  inv: self.account.balance >= -JuniorCard.juniorBankLine
  inv: self.bankLine >= 0
```

implicam os invariantes de seu super-tipo, `BasicCreditCard`:

```
context BasicCreditCard
  inv: self.account.balance >= -self.bankLine
  inv: self.bankLine >= 0
```

Seja uma operação que ocorra em uma classe "classe1", com pré-condição "pre1" e pós-condição "pos1", assim como sua sub-classe "classe2", com pré-condição "pre2" e pós-condição "pos2". A sub-tipagem comportamental (ou *Princípio de Liskov*) requer que sejam válidas as implicações $pre1 \rightarrow pre2$ e $pos2 \rightarrow pos1$ ("As pré-condições podem ser enfraquecidas; as pós-condições, fortalecidas."). No exemplo a seguir:

```
context BasicCreditCard::debit(sum:Integer)
  pre: debitPermit(sum)
  post: self.account.balance = self.account.balance@pre - sum
  pre: not debitPermit(sum)
  post: self.account.balance = self.account.balance@pre
  pre: true
  post: self.bankLine = self.bankLine@pre

context BonusCard::debit(sum:Integer)
  pre: debitPermit(sum)
  post: self.account.balance = self.account.balance@pre - sum and
        self.bonus = self.bonus@pre + BonusCard.bonus(sum)
  pre: not debitPermit(sum)
  post: self.account.balance = self.account.balance@pre and
        self.bonus = self.bonus@pre
  pre: true
  post: self.bankLine = self.bankLine@pre
```

[5]Esta abordagem viola o Princípio de Liskov. Para uma discussão mais localizada, vide final da Seção 2.4

as pré-condições são mantidas inalteradas, mas as duas primeiras pós-condições são fortalecidas na sub-classe `BonusCard`: caso seja permitido o uso do cartão, deve-se acrescentar um bônus ao total já acumulado nele; caso não seja permitido, o bônus acumulado deve estar inalterado ao final da operação.

3.8 Ocl4Java

Na Universidade de Kent, Reino Unido, foi desenvolvida a ferramenta *Ocl4Java* [AP04, AL03]. Parte do projeto *KMF* (Kent Modeling Framework), ela faz a geração de código Java a partir de expressões OCL presentes em diagramas de classe. Suporta a UML 1.4 e OCL 2.0.

O processamento feito pela ferramenta é dividido em duas etapas: análise (léxica, sintática e semântica) e síntese (geração de código ou interpretação). Ao fim da análise sintática, é gerada uma AST das expressões OCL. O analisador semântico consome essa AST, juntamente com o diagrama de classe de um dado modelo UML (referido pelas expressões OCL), produzindo um *modelo semântico*, que pode ser usado para a geração de código Java correspondente às expressões fornecidas.

Os autores identificaram que a gramática fornecida na especificação da OCL [OCL03] é ambígua. Criando *regras de remoção de ambigüidades* (disambiguating rules), desenvolveram uma gramática LALR para sua implementação (vide Anexo A do trabalho dos autores [AP04]). Para a análise léxica, essa gramática foi particionada em dois níveis: o primeiro contém os termos básicos da linguagem; o segundo contém as sub-gramáticas associadas a cada um dos termos definidos no primeiro nível. Isso foi feito porque, segundo os autores, permite que a gramática da linguagem seja descrita utilizando apenas seus símbolos básicos. Para a construção do analisador léxico, foi utilizado o *JFlex*, um gerador de analisadores léxicos. Como representação intermediária da conversão das expressões OCL, foi escolhida a AST, dada "sua habilidade de capturar toda informação necessária à execução de quaisquer operações relacionadas ao fluxo de controle e de dados do código-fonte".

A interação do analisador semântico com o diagrama de classe UML é feita utilizando-se uma *ponte* (bridge), entre as abstrações e as implementações dependentes do modelo. Classes que pertençam ao meta-modelo da UML são redefinidas para que sejam membros de um único pacote, sendo a implementação baseada nos tipos Java encontrados nos pacotes *java.lang* e *java.util*.

Os autores apresentam um exemplo de conversão de OCL para Java [AL03]). A Figura 3.4 mostra um possível diagrama de classe, a partir do qual supomos ter sido construída a restrição apresentada. A expressão OCL do exemplo é a seguinte:

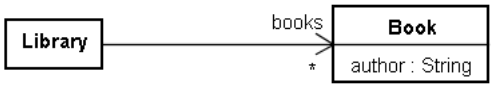

Figura 3.4: Diagrama de classe utilizado no exemplo de uso da Ocl4Java.

```
context library::Library
  inv: self.books->asSequence()->first().author
```

Dado o conjunto de livros (`self.books`) de uma biblioteca (`context library::Library`)[6], colocá-los em seqüência numa determinada ordem (`->asSequence()`), obtendo o primeiro livro dessa seqüência (`->first()`). Desse livro, identifique o autor (`.author`). Vale ressaltar que o resultado dessa expressão não é booleano, retornando um valor do tipo de `author` (dado o diagrama, `String`). Sendo assim, ela não é um invariante. Provavelmente, esse exemplo faz parte de uma expressão maior, que, de fato, é um invariante da classe `Library`. Essa expressão OCL é assim convertida para Java:

```
try {
  // Call property 'books'
  OclSet t17 = StdLibAdapterImpl.INSTANCE.Set(self.getBooks());
  // Call operation 'asSequence'
  OclSequence t16 = (OclSequence)t17.asSequence();
  // Call operation 'first'
  library.Book t15 = (library.Book)t16.first();
  // Call property 'author'
  library.Author t14 = (library.Author)t15.getAuthor();
  // return result
  if (t14 != null) return t14;
  else return StdLibAdapterImpl.INSTANCE.Undefined();
} catch (Exception e) {
  return StdLibAdapterImpl.INSTANCE.Undefined();
}
```

3.9 Comparação das abordagens

As abordagens estudadas encontram-se sumarizadas na Tabela 3.1. Para os casos em que uma dada informação sobre a ferramenta em questão não esteve clara, foi utilizado o valor "n/i" (não informado). A descrição de cada coluna segue abaixo:

Fluxos: Fluxo técnico do Praxis [Fil03], onde a atuação da ferramenta é recomendada. Caso uma dada ferramenta auxilie na modelagem de conceitos relevantes ao domínio do problema em um projeto de desenvolvimento de software, então julgou-se adequada para o fluxo de *Análise*. Se a ferramenta atua na modelagem da solução de um software, auxiliando na definição de uma estrutura implementável, considerou-se adequada ao fluxo de *Desenho*. Por outro lado, caso a ferramenta possa ser utilizada como auxiliar na geração de código e implementação do desenho de um projeto de software, considerou-se adequada para o fluxo de *Implementação*. Por fim, se a ferramenta auxiliar na detecção e remoção de defeitos, considerou-se adequada para o fluxo de *Testes*. Os fluxos *Requisitos* e *Engenharia de sistemas* não foram considerados.

Suporte: Tipo de suporte oferecido pela ferramenta, adaptado da caracterização feita por Hussmann e outros [HDF00]: *Análise sintática*, *Verificação de tipos*, *Consistência lógica* (identificação de contradições num dado conjunto de restrições), *Validação dinâmica* (avaliação de

[6] `library` é o nome do pacote dentro do qual estão as classes do diagrama.

invariantes, pré e pós-condições em tempo de execução; geralmente por meio de código Java gerado e inserido no código-fonte Java a ser testado), *Automação de testes* (verificação automática de resultados de testes, usando pré e pós-condições OCL para derivar casos de teste), *Verificação de código* (verificação de conformidade do código-fonte às restrições presentes no diagrama de classe respectivo).

Automação: Mostra se a operação da ferramenta não requer nem possibilita a interação do usuário. Nesse caso, ela é *Automática*; caso contrário, *Semi-automática*.

Versão UML: Versão da UML suportada pela ferramenta.

Versão OCL: Versão da OCL suportada pela ferramenta.

Versão Java: Versão de Java suportada e/ou produzida pela ferramenta. Curiosamente, nos trabalhos analisados, dificilmente é mencionada a versão suportada.

Parser XMI: Versão do analisador sintático XMI, que a ferramenta utilize para ler os diagramas de classe; caso contrário, é informado *n/d* (não disponível).

3.10 Mapeamentos entre OCL e Java

Analisando as diferentes ferramentas de engenharia direta, foi possível identificar, em linhas gerais, possíveis mapeamentos entre OCL e Java, em relação a seus tipos e operações. Esses mapeamentos se encontram compilados abaixo. A hierarquia de pacotes das classes Java será prefixada ao nome da interface ou classe apenas quando se julgar necessário para melhor entendimento.

Os tipos primitivos de OCL foram mapeados em tipos primitivos e classes embrulhadoras em Java (exceção feita ao tipo OCL `String`, mapeado para a classe `java.lang.String`), conforme a Tabela 3.2. Para todos os tipos, o operador relacional de igualdade de OCL ("=") é mapeado em Java para o operador "==", quando os objetos dos tipos Java envolvidos na operação forem primitivos; quando a operação envolver uma classe embrulhadora, utiliza-se o método `equals()`. A maior parte das operações do tipo OCL `String` são completamente suportadas pelos métodos da classe Java homônima, embora esses métodos possam ter nomes diferentes (por exemplo, OCL: `size()`; Java: `length()`). O restante das operações envolve a conversão do valor numérico a partir de uma String – `toInteger()` e `toReal()`. Essas operações são implementadas utilizando os métodos estáticos `valueOf()` das classes embrulhadoras `Integer` (ou `Long`) e `Float` (ou `Double`), respectivamente. Para os tipos numéricos OCL, `Integer` e `Real`, a maioria das operações são mapeadas diretamente em Java. O restante – `abs()`, `max()` e `min()` – é implementado em Java, usando os métodos estáticos de mesmo nome da classe `java.lang.Math`. Por fim, o tipo OCL `Boolean` é completamente implementado em Java. Suas operações são implementadas pelos operadores disponíveis em Java ou por uma composição deles – por exemplo, em OCL: `self implies b`; em Java: `(this ? b : false)`, ou na forma `(!this) || (this && b)`.

Os tipos coleção OCL foram mapeados para interfaces Java pertencentes ao *Arcabouço de Coleções* (Collections Framework) Java ou para classes que implementem direta ou indiretamente essas interfaces (vide Tabela 3.3). A maior parte das operações dos tipos OCL `Collection`, `Set`, `Bag` e `Sequence` é suportada pelas interfaces Java correspondentes. Entretanto, expressões de iteração são implementadas na forma de laços, nos quais a condição fornecida como parâmetro é avaliada para

Tabela 3.1: Sumário das ferramentas de engenharia direta

Ferramenta	Fluxos	Suporte	Automação	Versão UML	Versão OCL	Versão Java	Parser XMI
USE	Análise, Testes	Análise sintática, Verificação de tipos, Consistência lógica, Validação dinâmica	Semi-automática	2.0	2.0	n/i	n/d
DOT	Implementação, Testes	Análise sintática, Verificação de tipos, Validação dinâmica	Automática	1.4	2.0	n/i	n/i
Verheecke	Desenho, Implementação, Testes	Análise sintática, Verificação de tipos, Validação dinâmica	Semi-automática	1.3	1.3	n/i	n/i
SPACES	Testes	Análise sintática, Automação de testes	Semi-automática	n/i	n/i	n/i	1.2
ocl2j	Testes	Análise sintática, Verificação de tipos, Validação dinâmica	Automática	n/i	2.0	1.4	n/d
KeY Tool	Análise, Desenho, Implementação, Testes	Análise sintática, Verificação de tipos, Consistência lógica, Validação dinâmica, Verificação de código	Semi-automática	n/i	n/i	n/i	n/i
Ocl4Java	Implementação, Testes	Análise sintática, Verificação de tipos, Validação dinâmica	Automática	1.4	2.0	n/i	n/i

Tabela 3.2: Mapeamentos de tipos primitivos de OCL para Java

Tipo OCL	Primitivos Java	Classes embrulhadoras Java
`Integer`	`int, long`	`Integer, Long`
`Real`	`double, float`	`Double, Float`
`Boolean`	`boolean`	`Boolean`

cada elemento da coleção, iterativamente. De `Collection`, são: `iterate()`, `exists()`, `forAll()`, `isUnique()`, `one()` e `collect()`; de `Set`, `Bag` e `Sequence`, têm-se: `select()`, `reject()`, `collect()`, `collectNested()` e `sortedBy()`.

As tuplas foram implementadas em Java diferentemente das coleções. Para cada combinação de seus campos constituintes, criou-se uma classe Java respectiva. Por exemplo, sejam os seguintes literais de tuplas em OCL:

```
Tuple{idade:Integer=14, nome:String='João'}
Tuple{peso:Real=40.5, altura:Real=1.5, sexo:String='masculino'}
```

Em Java, para armazenar esses literais, seriam criadas classes contendo campos de tipos respectivos, que seriam instanciadas da seguinte maneira:

```
new TupleType_Integer_String(29, "João")
new TupleType_Real_Real_String(40.5, 1.5, "masculino")
```

O tipo `OclVoid` foi mapeado para `void`, em Java[7], e sua única instância, `OclUndefined`, foi mapeada para `null`. O tipo OCL `OclAny`, super-tipo de todos os tipos no modelo UML e dos tipos primitivos da biblioteca padrão da OCL, foi mapeado para a classe `java.lang.Object`. A implementação das operações de igualdade e diferença utilizou o método `equals()` (e sua negação, respectivamente). A operação de verificação se um objeto é de um dado tipo OCL, `oclIsTypeOf(t:Tipo)`, é implementada em Java comparando-se a saída do método `getClass()`, de `java.lang.Object`, com o que é produzido pelo nome da classe, sufixado por `.class`. A verificação se um objeto de um dado tipo está em conformidade com outro (i.e., ele é do mesmo tipo informado ou é de um de seus sub-tipos), `oclIsKindOf(t:Tipo)`, é feita utilizando-se o operador Java "`instance of`". As operações `oclIsNew()`, `oclIsInState()` e `allInstances()` apresentam dificuldades para seu mapeamento em Java, mostradas na Seção 3.11.

A palavra-chave OCL `self`, que se refere a um objeto da classe do contexto, é mapeada em Java para a palavra-chave `this`. A palavra-chave OCL `let` permite a definição de sub-expressões, possibilitando sua reutilização numa mesma restrição. Similarmente, a palavra-chave OCL `def` possibilita essa reutilização em todas as restrições de um dada classe contextual. Ambas construções foram implementadas em Java na forma de métodos de consulta (query methods) nas classes respectivas, sendo um método para cada ocorrência de `let` e de `def` no modelo UML. A palavra-chave OCL

[7] O mapeamento de `OclVoid` para Java apresenta dificuldades, conforme apresentado na Seção 3.11

Tabela 3.3: Mapeamentos de tipos coleção de OCL para Java

Tipo OCL	Interface Java
`Collection`	`java.util.Collection`
`Set`	`java.util.Set`
`OrderedSet`	`java.util.List`
`Bag`	`java.util.List`
`Sequence`	`java.util.List`

`body` é utilizada para indicar que a expressão associada a ela produz o resultado de uma operação de consulta. Em Java, ela corresponde à própria implementação da operação.

3.11 Problemas identificados

Diferentes autores identificaram dificuldades na execução de engenharia direta de diagramas UML – anotados com expressões OCL – para Java. Suas observações a esse respeito encontram-se compiladas abaixo, divididas em cinco categorias: falta de mecanismos explícitos em Java; definição incompleta de operações e de tipos OCL; problemas com referência a valores anteriores; problemas com a especificação de OCL e sua gramática; dificuldades de compreensão de OCL.

Falta de mecanismos explícitos em Java. Akehurst e Linington [AP04], utilizando a versão 1.4 de Java, identificaram recursos fornecidos por UML e OCL cuja engenharia direta para a linguagem Java apresenta grandes dificuldades: mecanismo explícito para a criação de enumerações; e classes de coleção tipadas (em sua versão 1.5, Java suporta ambos os conceitos: no primeiro caso, por meio da declaração `enum`; no segundo, por meio de tipos parametrizados ou *Generics*). Briand e outros [BDL04, DBL05] encontraram outros problemas. Em OCL, as operações `oclInState(s:OclState)`, `allInstances()` e `oclIsNew()` estão pré-definidas para quase todos os objetos, pois estão definidas para o tipo `OclAny` (super-tipo de quase todos os tipos em OCL). A operação `oclInState(s:OclState)` retorna `true` se o objeto chamador estiver no estado `s` (em OCL, é uma enumeração), conforme especificado no *diagrama de estados* (statechart) associado a essa classe. A dificuldade, segundo os autores, é que Java 1.4 não possui uma construção específica para enumerações (problema sanado na versão 1.5), tornando a conversão OCL específica para uma determinada implementação, dificultando a automação dessa conversão. A operação `allInstances()` retorna o conjunto de todas as instâncias de um dado tipo. Segundo os autores, para implementá-la, os construtores das classes deveriam ser alterados para que, a cada criação de um objeto, uma referência a ele deveria ser adicionada à coleção de objetos daquela classe. Caso o código-fonte não esteja disponível (por exemplo, componentes adquiridos de terceiros), essa implementação não é possível. Já `oclIsNew()` é apenas usada em pós-condições, retornando `true` caso o objeto a partir do qual é chamada tenha sido criado durante a execução da operação. Para essa operação, é necessário alterar o método respectivo, criando e mantendo uma coleção de todos os objetos criados dentro de seu escopo. Ao fim da execução do método, é feita a busca, nessa coleção, pelo objeto que chamou a operação `oclIsNew()`. Como no caso de `allInstances()`, é necessário que o código-fonte esteja disponível. Para implementar essas operações, os autores utilizaram POA, por meio de AspectJ.

Definição incompleta de operações e de tipos OCL. Akehurst e Patrascoiu [AP04] explicitaram um problema em relação à implementação do tipo `OclVoid`. Segundo a hierarquia de tipos da biblioteca padrão da OCL, extraída de sua especificação [OCL03] e representada na Figura 2.2, esse tipo estende todos os tipos da hierarquia, inclusive os tipos de coleção (i.e., `Collection`, `Set`, `Sequence`, `Bag` e `OrderedSet`). Entretanto, alguns desses tipos possuem operações com a mesma assinatura, mas diferentes tipos de retorno. Identificamos oito diferentes conflitos, apresentados abaixo no seguinte formato: <nome e parâmetros da operação conflituosa>: <tipo 1, onde há a

operação conflituosa> (<tipo de retorno de tipo 1>), <tipo 2, onde há a operação conflituosa> (<tipo de retorno de tipo 2>), e assim por diante.

1. `union(s:Set(T))`: `Set` (`Set(T)`), `Bag` (`Bag(T)`);
2. `intersection(bag:Bag(T))`: `Set` (`Set(T)`), `Bag(T)` (`Bag(T)`);
3. `excluding(object:T)`: `Set` (`Set(T)`), `Bag` (`Bag(T)`), `Sequence` (`Sequence(T)`);
4. `flatten()`: `Set` (`Set(T)`), `Bag` (`Bag(T)`), `Sequence` (`Sequence(T)`);
5. `including(object:T)`: `Set` (`Set(T)`), `Bag` (`Bag(T)`), `Sequence` (`Sequence(T)`);
6. `append(object:T)`: `OrderedSet` (`OrderedSet(T)`), `Sequence` (`Sequence(T)`);
7. `insertAt(index:Integer,object:T)`: `OrderedSet` (`OrderedSet(T)`), `Sequence` (`Sequence(T)`);
8. `prepend(object:T)`: `OrderedSet` (`OrderedSet(T)`), `Sequence` (`Sequence(T)`).

Caso o tipo `OclVoid` seja utilizado, o usuário não pode valer-se da hierarquia para usar todas as operações, em vista dos conflitos de herança existentes[8]. Em outra questão, a análise sintática de expressões da forma `expr.oclAsTypeOf(Type)`[9] é abordada. Ela falhará caso seja fornecido como argumento um tipo tupla ou coleção, pois não são sub-tipos de `OclAny` (vide Figura 2.2). Segundo os autores, a definição de expressões literais deveria ser estendida para incluir esses tipos. De fato, vão mais longe nessa afirmação, não vendo "qualquer razão para que coleções e tuplas não sejam consideradas sub-tipos de `OclAny`" [AP04]. Em outro trabalho, Baar [Baa05] apresenta um estudo sobre a falta de entendimento a respeito das expressões não-determinísticas presentes na linguagem OCL. "Elas são completamente ignoradas na literatura sobre OCL, sua semântica disponível na descrição oficial da linguagem OCL é mal-definida e nenhuma das ferramentas de OCL atuais suportam essas expressões de modo consistente". Ele aborda a operação `any(iterator:Object|body:Boolean)` – de `Collection`. Segundo levantamento do autor, no meta-modelo UML, que foi elaborado por alguns dos maiores especialistas em UML, não há uma única restrição verdadeiramente não-determinística presente (`any` é utilizada apenas em conjuntos contendo um único elemento, operando deterministicamente). Isso parece evidenciar uma certa falta de "familiaridade" com essa operação, mesmo por parte de especialistas da área. Segundo o autor, `any()` (e demais operações não-determinísticas) deveria ser usada quando o desenhista de software, deliberadamente, deseja sub-especificar uma restrição, oferecendo maior liberdade para o implementador. Hussmann e Zschaler [HZ04] apontam problemas relacionados ao uso de operações de mensagens da linguagem OCL. O primeiro diz respeito à impossibilidade do uso de variáveis livres (free variables) em expressões de mensagens: é impossível expressar que uma operação tenha sido chamada com um parâmetro cujo valor tenha sido menor que 10, por exemplo. A ordem das mensagens enviadas por um dado objeto também não pode ser especificada, pois OCL não permite comparar os tempos relativos entre os envios de mensagens. Além disso, não encontra-se claro na especificação o que deve ocorrer caso uma mensagem seja enviada mais que uma vez durante o tempo entre as avaliações das pré e pós-condições.

[8] Em C++, por exemplo, o código-fonte até será compilado, mas durante a execução, ocorrerá um erro ao se tentar usar uma operação conflituosa, definida em super-tipos diferentes.

[9] O nome para a operação está incorreto. Os autores provavelmente pretenderam abordar `oclIsTypeOf` – que verifica se `expr` é de um dado tipo – ou `oclAsType` – que faz o elencamento (`casting`) de `expr` para o tipo fornecido como argumento.

Problemas com referência a valores anteriores. Outro problema diz respeito a expressões que referenciem valores anteriores em pós-condições, utilizando o operador `@pre`. Nas ferramentas de engenharia direta que suportam a conversão desse operador (por exemplo, ocl2j), mostrou-se necessário clonar cada objeto referenciado. Isso é agravado no caso de coleções, onde cada um de seus elementos deve ser clonado, tornando cara a avaliação de pós-condições desse tipo em coleções grandes. Além disso, o uso desse operador pode levar a expressões cuja computação possa ser muito cara. Por exemplo, seja a expressão `self.b.c@pre`, parte de uma pós-condição. Ela deveria produzir o valor de `c`, anterior à execução da operação. Entretanto, o valor de `b` a ser usado é aquele resultante ao fim da execução da operação, quando a pós-condição é avaliada: "Antes da invocação do método, não é conhecido o valor futuro que a propriedade `b` assumirá, e então não é possível armazenar o valor de `self.b.c@pre` para uso posterior na pós-condição!"[DBL05]. Neste caso, seria necessário armazenar todos os possíveis valores de `b`, de antes da execução da operação, para selecionar o valor adequado posteriormente.

Problemas com a especificação de OCL e sua gramática. Akehurst e Linington [AL03] afirmam que a especificação não deveria ser orientada por comportamento, mas sim, por implementação, pois julgam que OCL não é uma meta-linguagem adequada para definir o significado de outras linguagens. Segundo eles, é preferível definir uma máquina virtual e especificar seu comportamento utilizando linguagem natural (abordagem utilizada por C++, Java e outras linguagens). Akehurst e Patrascoiu [AP04] constataram que a gramática de OCL não é adequada para gerar um analisador sintático, pois é ambígua. A gramática utiliza informação contextual para desambiguar (disambiguate), informação esta "que não se encontra disponível durante uma análise puramente baseada em sintaxe". Os autores também encontraram, ao longo da especificação, chamadas a operações que não se encontram definidas em nenhuma parte da especificação (por exemplo, na pág. 90, operação `tail`, no objeto `names`, tipo `Sequence`). Não necessariamente um problema, mas uma característica da linguagem, foi apontada por Baar [Baa05] (e também por Devos e Steegmans [DS05]). OCL é uma linguagem restritiva: suas pós-condições apenas restringem o conjunto de pós-estados possíveis. Em OCL, não é possível descrever como esses pós-estados são construídos. Dessa forma, ao fim de uma operação, qualquer estado que não contradiga as pós-condições é considerado válido, mesmo alterações em atributos que nada se relacionam com a operação anotada com a restrição! Segundo Markovic e Baar [MB05], "uma formalização em OCL das regras deve ser lida como 'tudo pode mudar, a menos que seja explicitamente indicado que o valor inicial deva permanecer inalterado' ". Para ilustrar isso, vide o exemplo representado na Figura 3.5. A pós-condição da operação

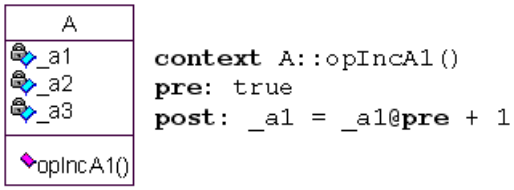

Figura 3.5: Exemplo de pós-condição sub-especificada

`opIncA1()` informa que, ao final de sua execução, o valor do atributo `_a1` deve ser incrementado.

Entretanto, essa pós-condição nada instrui quanto ao valor das demais variáveis: caso a implementação dessa operação alterasse os valores de `_a2` e `_a3`, seria considerada uma implementação válida. Para se obter a precisão desejada, é necessário explicitar que todos os demais atributos permanecem inalterados, acrescentando as pós-condições: `_a2=_a2@pre` e `_a3=_a3@pre`. Para classes maiores e mais complexas, isso pode resultar num grande aumento do tamanho da especificação UML que se pretende descrever com maior precisão. Na literatura, esse problema é conhecido como o *Problema de Moldura* (Frame Problem), definido por McCarthy e Hayes [MH69]. Outro problema refere-se à definição da operação `commonSuperType` na meta-classe UML `Classifier`. Essa operação é usada durante o processo de inferência de tipos da especificação OCL 2.0 [OCL03] (vide seção 8.3.8). Por exemplo: `Set{0, 1, 3.5}`. O tipo desse literal é inferido como sendo `Set{Real}`. No caso de classes com herança múltipla que herdem de mais de um ancestral comum, a especificação não define o que deve ocorrer. Por exemplo, vide a Figura 3.6, extraída do trabalho de Hussmann e Zschaler [HZ04]. As regras de boa formação presentes na especificação não esclarecem qual seria o tipo inferido de `Set{Studiyng_Politician,Political_Student}`. Os autores também apontam

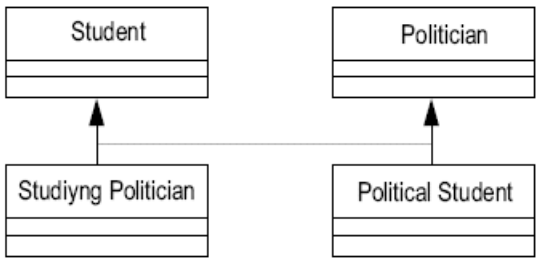

Figura 3.6: Exemplo de problema de inferência em herança múltipla

a impossibilidade de se derivar diretamente um analisador sintático a partir da especificação da linguagem, dado que a gramática se encontra na forma de um meta-modelo MOF [HZ04]. Ainda a respeito da especificação de OCL, Ahrendt e outros [ABB+05] também afirmam que ela não é precisa quanto ao significado de múltiplas pré e pós-condições. Em algumas abordagens, realiza-se a conjunção de todas as pré-condições para formar uma única pré-condição global (o mesmo é feito para pós-condições); em outras, cada par é verificado separadamente dos demais. Por fim, em uma análise feita pela empresa EmPowerTec, dentre as 45 expressões de exemplo presentes na especificação da OCL, foram encontradas 20 incorretas [EmP05].

Dificuldades de compreensão de OCL. Segundo a especificação da OCL [OCL03], "é uma linguagem formal que é fácil de ler e escrever". Porém, Siikarla e outros [SPS04] constatam que, dada sua experiência, a afirmação citada está mais próxima da definição de uma meta do que da constatação de um fato. Segundo eles, o contato inicial com a linguagem tende a confundir os desenvolvedores; expressões curtas e diretas são, freqüentemente, muito simples, mas quando sua complexidade e tamanho aumentam, a clareza e legibilidade diminuem radicalmente. Além disso, de acordo com Kollmann e Gogolla [KG01a], o uso de expressões OCL em diagramas UML confronta-se com a idéia original de tornar mais fácil o entendimento da arquitetura subjacente de um software por meio de uma representação gráfica, pois "o leitor ainda precisa entender as regras dadas em um tipo de linguagem de programação".

Capítulo 4

Estado-da-arte em Engenharia Reversa

O desenvolvimento de software envolve um grande número de artefatos, cuja consistência entre eles deve ser mantida. Um dos processos auxiliares nessa tarefa é a engenharia reversa. Neste trabalho, o objetivo foi levantar o estado-da-arte de ferramentas de engenharia reversa de softwares Java para diagramas de classe da UML, e verificar a viabilidade de se empregar OCL nesse processo para não perder informações sobre a implementação escolhida pelo desenvolvedor.

4.1 O que é Engenharia Reversa?

O desenvolvimento de software envolve um grande número de artefatos de software (por exemplo, especificações de requisitos, modelo de análise, modelo de desenho, código-fonte, executáveis). Todos esses artefatos, embora tratem do mesmo produto, o apresentam sob diferentes enfoques. Alterações realizadas num artefato podem referir-se a conteúdo presente nos demais, sendo importante manter a consistência entre esses artefatos. Um dos mecanismos usados para assegurar essa consistência, pelo menos entre o código (seja ele fonte ou executável) e o modelo de desenho, é a engenharia reversa.

Segundo Chung e Lee [CL01], "engenharia reversa de software é um processo usado para criar um modelo a partir de um determinado código, utilizando algum mapeamento de uma linguagem de implementação específica". Ela lida com a necessidade de se compreender um sistema existente, recuperando seu modelo de desenho correspondente.

No contexto de desenvolvimento de software, engenharia reversa é o processo que, utilizando como insumo o código-fonte de um programa, obtém sua representação em nível mais alto de abstração, filtrando as informações desejadas. As informações recuperadas devem refletir a implementação do sistema em questão, fornecendo (semi-) automaticamente a mesma informação que seria recuperada manualmente pelos desenvolvedores.

Além de ser usada em desenvolvimento de software, a engenharia reversa também é empregada em outros contextos, tal como em bancos de dados relacionais. Em seu trabalho sobre recuperação de associações n-árias, Soutou [Sou97] apresenta várias razões para aplicação de engenharia reversa: "manutenção e redesenho (maior precisão na criação de um novo esquema, a partir de um

já existente), migração de dados (de bancos de dados relacionais para especificações orientadas por objeto), integração de bancos de dados (geralmente, realizadas em nível conceitual)".

4.2 Por que é importante?

Softwares precisam evoluir. À medida que um sistema é expandido e alterado para atender a novos requisitos, o código se torna cada vez mais complexo, afastando-se do seu desenho original e diminuindo sua qualidade interna (por exemplo, analisabilidade, modificabilidade, testabilidade). Surpreendentemente, métodos e ferramentas melhores e mais avançados não resolvem esse problema. Seus novos recursos são usados para implementar mais requisitos, no mesmo intervalo de tempo, tornando o software cada vez mais complexo [GSMD03]. A engenharia reversa é uma das ferramentas que pode ser usada para enfrentar essa *espiral de complexidade*, mantendo a consistência entre o código-fonte do sistema e seu modelo de desenho.

Guéhéneuc e Albin-Amiot [GAA04] afirmam que "a recuperação automatizada de modelos de desenho a partir da implementação de um sistema é um problema recorrente na comunidade de (re-) engenharia de software orientada por objetos". Modelos de desenho tornam-se freqüentemente obsoletos durante a implementação, apresentando grandes discrepâncias em relação ao código respectivo implementado. Tais modelos podem não ser de grande auxílio para engenheiros de software – sejam desenvolvedores, sejam mantenedores –, que devem entender ou analisar o sistema durante seu desenvolvimento e, posteriormente, durante sua manutenção [GAA04]. Se uma implementação estiver em conformidade com seus requisitos, mas divergir do desenho documentado, sua manutenibilidade pode estar comprometida. Para minimizar esse problema, um modelo gerado por engenharia reversa pode auxiliar na identificação de inconsistências entre o modelo planejado e sua implementação efetiva. Cooper e outros [CKvKR04] declaram que isso "pode evitar dificuldades posteriores durante a fase de manutenção do produto, assegurando a consistência entre todos os artefatos produzidos durante o desenvolvimento de software". Segundo Briand e outros [BLM03], "técnicas de engenharia reversa são necessárias para entender a estrutura e o comportamento de um sistema de software cuja documentação esteja desatualizada ou indisponível, principalmente no contexto onde amarração dinâmica (dynamic binding) e polimorfismo são muito usados (como em Java)". Dessa forma, ferramentas automatizadas para recuperar modelos de desenho a partir do código (fonte ou executável) produzido de uma aplicação são úteis, pois o código pode ser a única fonte de dados disponível durante sua fase de manutenção.

4.3 Escopo do trabalho

Nesse trabalho, foram levantadas ferramentas de engenharia reversa que atuam sobre código Java, em código-fonte e/ou bytecodes, gerando o modelo respectivo em diagramas de classe de um modelo UML. Dentre os elementos desses diagramas recuperados pelas ferramentas, priorizou-se o estudo da recuperação dos seguintes: classes, interfaces, associações binárias simples, agregações, composições, realizações, generalizações e dependências. Conforme discutido, a escolha por Java se devem a maior

experiência com a linguagem e farta documentação disponível, além de ser uma linguagem moderna e poderosa, e que oferece muitos recursos.

Alguns trabalhos sobre engenharia reversa focam na recuperação de desenho a partir da identificação de mecanismos de desenho [GHJV94] presentes no código gerado [SS03b, NNZ00, SvG98]. Arcelli e outros [AMRT05] declaram que

> "O reconhecimento desses mecanismos pode fornecer informação adicional relacionada à lógica por trás do desenho implementado. Além de fornecer informação sobre como a arquitetura do sistema foi construída, eles indicam os motivos pelos quais ela foi construída de determinada forma, devido à semântica que mecanismos de desenho possuem. Contudo, a detecção desses mecanismos no processo de engenharia reversa encontra forte resistência entre as comunidades de pesquisa de mecanismos de desenho e de engenharia reversa. Isso se deve, fundamentalmente, às várias possíveis implementações, interpretações e objetivos no uso da mesma estrutura de um único mecanismo. Nesse contexto, há a necessidade de se descrever mais precisamente os mecanismos de desenho definidos, considerando-os como composições de elementos mais simples".

Os trabalhos abordados nas seções a seguir possuem diferentes objetivos com a realização de engenharia reversa. *Ptidej* [Gué04b] e *IDEA* [KG02] visam auxiliar mantenedores de software, por meio da redocumentação de programas Java usando a UML. Essas ferramentas recuperam informações desses programas, reproduzindo sua implementação em nível mais abstrato. *REVSA* [CKvKR04], *Womble* [JW99] e *class2uml* [Kes04] objetivam identificar eventuais inconsistências entre o modelo de desenho produzido durante o processo de desenvolvimento e o modelo recuperado por engenharia reversa da implementação efetiva desse modelo. *FUJABA* [NNZ00] se propõe a fornecer suporte para a engenharia de ida e volta entre código Java e UML. Na abordagem de *Seemann* [SvG98] e na ferramenta *SPQR* [SS03b], a engenharia reversa é utilizada para a identificação de mecanismos de desenho no código.

4.4 Ptidej

Ptidej (Pattern Trace Identification, Detection, and Enhancement in Java) [Gué04c, Gué04b, GAA04, GSZ04, Gué05, Gué04a] é um conjunto de ferramentas sendo desenvolvido na Universidade de Montreal, Canadá. Dentre outras funcionalidades, é capaz de realizar a engenharia reversa de programas Java para diagramas de classes UML 1.5, exibindo o resultado em sua interface gráfica.

O processo de criação de um diagrama de classe, semi-automático (utiliza interação com o usuário), se decompõe em 2 fases:

- Diagramas de classe são construídos a partir da análise estática dos arquivos de classes do programa (bytecodes), utilizando ferramentas especializadas, sendo descritos segundo um metamodelo próprio.
- Realiza-se o refinamento dos diagramas construídos, analisando-se o comportamento do programa em execução. Para isso, é utilizada outra ferramenta da suíte Ptidej: *Caffeine* [GDJ02].

Ptidej faz análise estática e dinâmica de programas Java, com o propósito de inferir relacionamentos binários entre suas classes e interfaces. É feita a engenharia reversa daqueles elementos

cuja recuperação, segundo definição usada pelo autor, seja *precisa* ("o quanto um dado valor está próximo de um valor padrão") e *abstrata* ("algo que concentra em si próprio as qualidades essenciais de algo mais amplo, mais geral ou de muitas coisas; essência") [Gué04c].

As associações identificadas pela ferramenta são unidirecionais. Ela interpreta que uma associação é bidirecional quando, entre duas classes, há duas associações (unidirecionais) em sentidos opostos. Relacionamentos de generalização são diretamente identificados analisando-se a hierarquia de classes do programa. Entretanto, para identificar os relacionamentos binários de *dependência de uso* (usage, palavra-chave «use») e associações, a ferramenta realiza uma análise dinâmica do programa. Segundo definições desses relacionamentos [GAA04], eles foram decompostos em 4 propriedades, usadas como base para a definição dos algoritmos de identificação dos relacionamentos:

Exclusividade (EX): Indica se uma instância de uma classe envolvida num relacionamento pode estar envolvida em outro relacionamento simultaneamente. É computada dinamicamente.

Sítio de invocação (IS): Indica quais instâncias de uma dada classe envia mensagens para instâncias de outra classe. É computada estaticamente.

Tempo de vida (LT): Corresponde ao tempo transcorrido entre os tempos de destruição de duas instâncias, pertencentes a duas classes distintas. É computada dinamicamente.

Multiplicidade (MU): Descreve o número de instâncias de uma classe que é permitido num relacionamento com uma instância de outra classe. É computada estaticamente.

Guéhéneuc e Albin-Amiot [GAA04] provam que esse conjunto de propriedades é mínimo e completo em relação às definições dos relacionamentos recuperados e das propriedades propostas na literatura. A correta detecção dos valores das propriedades computadas dinamicamente depende dos caminhos de execução do programa (limitação comum em análises dinâmicas).

Sejam duas classes "A" e "B" de um dado programa. É identificado um relacionamento de associação caso "A" envie uma mensagem para "B", analisando-se a propriedade MU(A,B). Se a definição de "A" – o todo – contiver instâncias de "B" – a parte –, o relacionamento binário inicialmente identificado como associação simples passa a ser considerado uma associação de agregação, analisando-se as propriedades IS e MU(A,B). Caso a instância do todo possua as instâncias de sua parte, sendo exclusivas dessa instância do todo e destruídas quando o todo é destruído, essa agregação passa a ser considerada uma composição, analisando-se as propriedades IS, MU, EX e LT(A,B)[1]. Quando não é nenhum dos casos acima, é inferido que o relacionamento é uma dependência de uso.

Em um de seus trabalhos, Guéhéneuc [Gué04c] apresenta uma lista de todos os elementos que podem estar presentes em diagramas de classe UML, conforme especificações da linguagem UML 1.5 [UML03], indicando quais desses elementos podem ser recuperados, segundo sua análise. Além isso, faz uma comparação do desempenho de recuperação da Ptidej com outras ferramentas de engenharia reversa de diagramas de classe.

[1] Segundo Milanova [Mil05], a definição de composição usada pelos autores da Ptidej, baseada em *posse exclusiva* (exclusive ownership), pode não ser suficiente para modelar mecanismos comumente usados, tais como: iteradores, decoradores e fábricas

4.5 REV-SA

REV-SA (Reverse Engineering and Verification using Static Analysis) [CKvKR04] foi outra ferramenta analisada. Desenvolvida na Universidade de Curtin, Austrália, objetiva identificar eventuais inconsistências entre um modelo de desenho e sua respectiva implementação em bytecodes Java. Isso é feito comparando-se os artefatos de desenho, produzidos por engenharia direta, com os artefatos recuperados por meio de engenharia reversa do código.

REV-SA faz a análise estática utilizando os arquivos de classe compilados (i.e., bytecodes), ao invés de usar o código-fonte. Isso foi feito para que a recuperação dos relacionamentos entre as classes seja possível mesmo quando o código-fonte não se encontre disponível. O processo é executado semi-automaticamente. Nos casos em que a ferramenta não é capaz de deduzir o relacionamento de modo preciso, as ambigüidades ou inconsistências encontradas são marcadas no diagrama de classe, para que o desenhista de software as remova posteriormente.

A ferramenta captura relacionamentos de generalização, realização e associação binária unidirecional (simples e de agregação). Segundo os autores, esses relacionamentos "são críticos para garantir que a implementação seja fiel ao desenho do sistema". Generalizações e realizações são descobertas extraindo-se as super-classes e a lista de interfaces implementadas por cada classe. Agregações são identificadas verificando-se as classes que contenham campos não-primitivos. A multiplicidade das agregações é recuperada por meio da inspeção de seus campos: por padrão, ela é '0..1'; caso haja mais que um campo do mesmo tipo, a multiplicidade é incrementada adequadamente; caso o campo seja um arranjo (array), ela é assumida como '0..*'. Composições não são identificadas, sendo consideradas como associações de agregação. Os autores declaram que a "distinção entre elas (agregação e composição) está perdida na implementação".

4.6 FUJABA

FUJABA (From UML to Java And Back Again) [NNZ00, NNWZ00] é uma ferramenta desenvolvida na Universidade de Paderborn, Alemanha. Seu objetivo é fornecer suporte para engenharia de ida e volta entre código Java e UML. Para isso, o código-fonte gerado deve ser modificado segundo certas regras impostas pela ferramenta, fazendo com que as alterações sejam replicadas nos modelos UML respectivos.

A análise de programas Java é feita estaticamente, a partir de seu código-fonte. São recuperadas classes, interfaces e os seguintes relacionamentos binários e unidirecionais: generalização, realização, associação, agregação e composição. Entretanto, Kollmann e outros [KSS+02] identificaram alguns problemas na recuperação de alguns desses relacionamentos: interfaces realizadas por classes são mostradas como se fossem herdadas; agregações e composições devem ser codificadas segundo formato específico da ferramenta; caso contrário, não são detectadas; para as associações, FUJABA não suporta restrições para os limites inferiores das multiplicidades.

Para auxiliar o processo de engenharia reversa, FUJABA procura identificar mecanismos de desenho [GHJV94] presentes no código-fonte. Segundo Arcelli e outros [AMRT05], partes comuns dos diferentes mecanismos de desenho são definidas separadamente e armazenadas num repositório

de *sub-mecanismos* (sub-patterns) – "elementos recorrentes e fundamentais que compõem os mecanismos de desenho". Posteriormente, esses sub-mecanismos são usados como elementos atômicos na definição de mecanismos de desenho, ou mesmo de outros sub-mecanismos. O algoritmo de detecção trabalha incrementalmente, individualizando inicialmente os sub-mecanismos, para depois combiná-los em mecanismos.

4.7 Womble

Womble [JW99] é uma ferramenta desenvolvida no MIT (Massachusetts Institute of Technology), Estados Unidos. Ela extrai, de modo automático, modelos de objetos a partir da análise de arquivos de classes Java. Como resultado, produz um modelo de objetos na forma de um grafo, onde os nós são as classes e interfaces Java identificadas, e arestas representam os relacionamentos recuperados: generalização, realização e associação simples (binária e unidirecional). Os resultados podem ser visualizados na própria ferramenta, mas também podem ser exportados para o software de visualização de grafos *DOT*, da AT&T Bell Labs. O modelo de objetos utilizado pela ferramenta segue uma notação própria, sendo um subconjunto aproximado da notação UML (versão não informada). Os autores optaram por usar como insumo para análise estática os arquivos de classes, pois "bytecodes contêm toda a informação presente no código-fonte, sendo mais compacto e mais disponível que ele".

Utilizando a Womble, o processo de engenharia reversa ocorre da seguinte forma. Para cada classe identificada, é acrescentada uma aresta entre ela e sua super-classe (generalização), e entre ela e todas as interfaces que implementa (realização). Associações unidirecionais são inferidas verificando-se, nas classes, a presença de campos cujos tipos sejam não-primitivos. As associações partem da classe que contém esses campos em direção às classes respectivas desses campos.

A inferência da multiplicidade das extremidades das associações é feita de maneira diferenciada para a ponta da seta (arrow head) de associação e para sua cauda (tail). A multiplicidade da ponta é inferida pela presença de arranjos ou de classes contêiner (container classes, como por exemplo, `java.util.Hashtable`) dentro da classe sendo analisada. Supõe-se, inicialmente, que não existam restrições para multiplicidade, ou seja, '0..*', e procura-se obter uma multiplicidade mais restrita. Se o campo não é um arranjo e nem referencia uma classe de coleção, sua multiplicidade pode ser limitada para '0..1' ou '1'. Se é atribuído ao campo o valor `null`, ou se a classe possui um construtor não-privado, dentro do qual não é atribuído nenhum valor ao campo, é escolhida a multiplicidade '0..1'. Caso contrário, é atribuído um valor ao campo, e, sendo assim, é escolhida a multiplicidade '1'. Para a cauda de uma associação, sua multiplicidade também inicia com '0..*'. Para explicar o mecanismo, sejam "A" e "B" duas classes, onde "A" possui uma associação com "B". Essa multiplicidade é restrita para '0..1', caso os objetos de "B" sejam campos privados em "A" e nenhum método de "A" retorne um objeto de "B". A ferramenta não tenta restringir essa multiplicidade para exatamente '1'.

4.8 IDEA

IDEA [KG01a, KG01b, KG02, KSS+02] é uma ferramenta de engenharia reversa que utiliza UML para redocumentar programas Java. Foca-se na análise estática das estruturas do código Java, reconhecendo os relacionamentos existentes e extraindo-os para diagramas de classe UML. Foi desenvolvida na Universidade de Bremen, Alemanha, dentro do projeto *UMLAID* (Abstract Implementation and Design with the Unified Modeling Language).

Para realizar o processo, IDEA utiliza um meta-modelo da linguagem Java, onde a estrutura estática do programa analisado é armazenada como um modelo (ou instância desse meta-modelo) [KG01b]. Essa análise é feita tendo como insumo os bytecodes do programa. Entretanto, o código-fonte deve estar presente, pois a ferramenta necessita compilá-lo, para que armazene informações de depuração em seus bytecodes. São realizadas transformações sucessivas nesse modelo até a criação de uma representação do desenho do programa.

A análise realizada pela IDEA é insuficiente em algumas situações, dada a ambigüidade ou complexidade do código, exigindo a interação com o usuário. Nesses casos, a ferramenta apresenta ao usuário as construções que recuperou, mas que não foi capaz de desambiguar.

Associações binárias unidirecionais são recuperadas analisando-se referências aos objetos presentes no código. São reconhecidas associações binárias bidirecionais se duas classes quaisquer possuírem, cada uma, um atributo da outra classe respectiva (o usuário pode auxiliar no reconhecimento). A determinação da multiplicidade das extremidades de uma associação é feita da seguinte forma: '0..1', se o elemento é iniciado (initialized) em posição desconhecida do código; '1', se o elemento é iniciado junto com a criação do objeto (no construtor ou no bloco de iniciação da classe); '*', se o elemento é uma classe de coleção (i.e., implementa a interface `java.util.Collection` ou uma de suas herdeiras).

Nomes de papéis (role names) das associações são derivados a partir dos identificadores de campos não-primitivos das classes, sendo mostrados próximos ao alvo de uma associação dirigida. Se houver classes de coleção no código, o tipo não-primitivo contido é representado no diagrama com a multiplicidade apropriada. Classes dos tipos `java.util.Vector`, `java.util.ArrayList` e arranjos são representados utilizando a notação de qualificador junto ao tipo contido (o atributo qualificado é seu índice) e '0..1' junto à extremidade-alvo da associação (target association end) (cada posição possui zero ou exatamente um elemento); a associação é anotada com o valor etiquetado `qualified`.

A recuperação de agregações é feita, primeiramente, identificando-se todos os campos de uma classe, cujos tipos sejam classes de coleção. A aceitação dessas agregações depende de certos parâmetros de configuração da IDEA. A forma mais simples de aceitação, embora imprecisa, é considerar agregação toda classe que contiver classes de coleção, onde o usuário pode validar manualmente as agregações candidatas. Em vista disso, os autores sugerem que a melhor forma de recuperar esse relacionamento é por meio de "conhecimento suficiente da arquitetura do software" [KSS+02].

Composições não são recuperadas, mas os autores sugerem propriedades delas que, quando possuídas por uma agregação, podem identificá-las: *posse forte* (strong ownership) requer que uma parte deva pertencer, no máximo, a um todo; *tempo de vida coincidente* (coincident lifetime) requer

que a iniciação de todas as partes não tenha ocorrido antes da iniciação do todo, além de seu ciclo de vida terminar, no máximo, quando o ciclo de vida do todo terminar.

4.9 class2uml

Outro método de engenharia reversa de código Java para diagramas de classe UML é apresentado por Keschenau [Kes04]. Sua ferramenta, *class2uml*, foi construída com o objetivo de auxiliar programadores a inspecionar o código-fonte de classes de um sistema, "comparando-se uma dada implementação com um documento de desenho existente". Foi desenvolvida na Universidade de Aachen, Alemanha.

A class2uml realiza análise estática sobre os bytecodes Java, construindo recursivamente o diagrama de classe respectivo a partir de uma dada classe, permitindo restringir o nível de profundidade de recursão e os prefixos de pacotes que devem ser investigados.

Os relacionamentos recuperados por sua abordagem são: associações (simples e de composição) e suas multiplicidades, generalizações, realizações e dependências. Também são identificados métodos de consulta presentes nas classes. Uma associação unidirecional entre duas classes é identificada quando um dado campo de uma classe for do tipo de outra classe. A multiplicidade da extremidade da associação (association end) é determinada pelo número de objetos da classe contida que o objeto da classe continente pode referenciar durante o tempo de vida deste objeto. Supõe-se, inicialmente, uma multiplicidade de '1', e procura-se obter uma multiplicidade menos restrita:

- Tipo do campo: se for um arranjo, a multiplicidade é '0..*'; caso contrário, mantém-se em '1'.
- Número de atribuições ao campo: '1' se houver apenas uma. Se houver mais que uma, '1..*'.
- Escopo de iniciação do campo: se iniciado em um construtor, '1'; em um método comum, '0..*'.
- Forma de iniciação do campo: se o campo for iniciado com o valor retornado por um método, sua multiplicidade é '0..1', pois não há como determinar se o método retornará uma referência ou `null`.

4.10 SPQR

Diferente das demais abordagens (exceto por FUJABA), *SPQR* (System for Pattern Query and Recognition) efetua engenharia reversa baseando-se no reconhecimento de mecanismos de desenho em código C++ [SS03b, SS03a] (Arcelli e outros [AMRT05] desenvolveram um protótipo baseado na SPQR que opera sobre código Java). É desenvolvida na Universidade da Carolina do Norte, Estados Unidos.

Em SPQR, mecanismos de desenho são definidos por meio dos seguintes elementos fundamentais:

EDPs (Elemental Design Patterns): mecanismos de desenho em nível mais baixo de abstração, que expressam um pequeno número de conceitos de orientação por objetos. Exemplos: herança (EDP *Inheritance*), delegação (EDP *Delegate*) [Smi02]. Os autores afirmam que, por meio da composição de EDPs, podem ser construídos todos os 23 mecanismos de desenho propostos pela Gang of Four [GHJV94].

Operadores de dependência (reliance operators): são expressões quantificáveis que informam se um elemento (objeto, método ou campo) depende da existência de outro para sua própria existência ou execução.

ρ-calculus: Vide trabalho de Smith e Stotts [SS03b].

A ferramenta realiza a detecção de mecanismos automaticamente, analisando a AST produzida pelo compilador. SPQR identifica os EDPs empregando suas definições em ρ-calculus. Utilizando como insumo os EDPs identificados e regras de inferência da ferramenta, um provador de teoremas detecta instâncias dos mecanismos de desenho e suas variações. Ao fim do processo, é produzido um relatório, em XML, com os mecanismos encontrados, que pode ser usado para a produção de diagramas UML.

Arcelli e outros [AMRT05] apresentaram um estudo comparativo entre FUJABA e SPQR, no qual avaliam o papel benéfico que a identificação de 'sub-mecanismos' desempenha durante o processo de engenharia reversa.

4.11 Abordagem de Seemann

Seemann e Gudenberg [SvG98] desenvolveram outra ferramenta de engenharia reversa, na Universidade de Würzburg, Alemanha. O objetivo de sua abordagem é o reconhecimento automático de mecanismos de desenho por meio de análise estática do código-fonte. A ferramenta proposta utiliza a notação da UML 1.1.

A ferramenta recupera os seguintes relacionamentos binários: generalização, realização, associações (simples e de agregação) unidirecionais e dependências de chamada (call). Generalização e realização são identificadas trivialmente, utilizando a hierarquia de herança de classes e interfaces, além das classes que implementem essas últimas (informação disponível diretamente no código). Dependências de chamada são identificadas quando um objeto de uma dada classe executa um método de um objeto de outra classe. Uma associação simples é identificada, quando um dos campos de uma classe referencia um objeto de outra classe, e executa métodos deste objeto. Essa associação será considerada uma agregação, se, adicionalmente, o objeto referenciado for criado pela própria classe continente. A multiplicidade de uma associação é obtida a partir do tipo dos campos. Para todos os campos de um dado tipo, é feita a interseção dos conjuntos de métodos executados desses campos: caso o resultado seja vazio, a multiplicidade é '1' em ambas as extremidades da associação (entre a classe continente e a classe referenciada); caso contrário, é '1' para a classe continente e '*' para a classe referenciada.

4.12 Comparação das abordagens

Na Tabela 4.1, é apresentado um sumário das ferramentas de engenharia reversa analisadas. Para os casos em que uma dada informação sobre a ferramenta em questão não esteve clara, será utilizado o valor 'n/i'. Vale ressaltar que não foram identificados trabalhos que anotassem, com expressões OCL, os diagramas de classe recuperados. Também, não foi encontrada, em nenhum dos trabalhos

analisados, a versão de Java suportada para a análise do código durante a engenharia reversa. A descrição de cada coluna segue abaixo:

Tipo de análise: Indica se a ferramenta utiliza apenas informações de tempo de compilação (*Estática*) ou se *informações de tempo de execução do software* (`trace`) são analisadas para recuperar os elementos UML (*Dinâmica*).

Formato do insumo: Caso a análise realizada pela ferramenta seja estática, indica se o insumo da análise é o código-fonte ou seus bytecodes.

Grau de automação: Mostra se o usuário pode auxiliar a ferramenta na identificação dos elementos recuperados (*Semi-automática*), por exemplo, removendo ambigüidades que ela não tenha sido capaz de remover, ou se toda a operação não requer interação com o usuário (*Automática*).

Exporta: Indica formato no qual a ferramenta pode exportar os resultados obtidos.

Elementos: Elementos recuperados. Todas as abordagens analisadas recuperam classes, interfaces e os relacionamentos de realização e generalização. Entretanto, no caso desta última, nenhuma das abordagens trata de restrições de conjuntos de generalização (i.e., `disjoint`, `overlapping`, `complete`, `incomplete`). Quando mencionado o termo *Simples*, indica-se a associação simples, binária e unidirecional (e suas respectivas multiplicidades). Quando for o caso, será indicado se a ferramenta também faz o reconhecimento de associações bidirecionais e/ou n-árias.

Versão UML: Versão da UML suportada pela ferramenta na geração do diagrama de classe. Nos trabalhos analisados, dificilmente é mencionada a versão suportada pela ferramenta.

Tabela 4.1: Sumário das ferramentas de engenharia reversa

Ferramenta	Tipo de análise	Formato do insumo	Grau de automação	Exporta	Elementos	Versão UML
Ptidej	Estática e dinâmica	Bytecodes e trace	Semi-automático	n/i	Simples, agregação e composição	1.5
REV-SA	Estática	Bytecodes	Semi-automático	XMI 1.1	Simples e agregação	n/i
FUJABA	Estática	Código-fonte	Automático	n/i	Simples, agregação e composição	n/i
Womble	Estática	Bytecodes	Automático	DOT	Simples	n/i
IDEA	Estática	Bytecodes	Semi-automático	n/i	Simples (n-ária e bidirecional) e agregação	n/i
class2uml	Estática	Bytecodes	n/i	n/i	Simples, composição e dependência	n/i
SPQR	Estática	AST	Automático	XML	n/i	n/i
Seemann	Estática	Código-fonte	Automático	n/i	Simples, agregação e dependência de chamada	1.1

4.13 Problemas identificados

No estado-da-arte de engenharia reversa de código Java para diagramas de classe UML, ainda existem dificuldades a serem superadas. Segue um levantamento desses problemas, segundo diversos autores:

1. Inexistência de processo. Engenharia direta é suportada por uma variedade de processos de desenvolvimento de software, tais como aqueles que utilizam os modelos de ciclo de vida em: cascata (com realimentação ou não), espiral, prototipagem evolutiva, entrega evolutiva [Fil03]. Entretanto, não foi encontrado um processo estabelecido na literatura para a engenharia reversa. No contexto de reengenharia, Mens e Tourwé [MT04] apostam no uso de *mecanismos de reengenharia*, que "podem fornecer soluções genéricas, baseadas em problemas recorrentes, encontrados em situações reais de modificação de softwares-legado".
2. Suporte heterogêneo à UML. As diferentes ferramentas existentes adotam extensões proprietárias da UML, o que dificulta o uso em diferentes fases de um mesmo projeto de software. Kollmann e outros se surpreendem em seu estudo [KSS+02] ao verificar que, nas ferramentas de engenharia reversa analisadas, não se tem usado todo o potencial da UML, mesmo que ela seja o padrão industrial de facto para apresentação de artefatos de desenvolvimento de software orientado por objetos.
3. Indefinição da abordagem a relações n-árias. Ainda há muita argumentação a respeito da modelagem de associações n-árias e binárias. Mesmo em modelagem de entidades-relacionamentos de bancos de dados relacionais, a extração automática de associações n-árias é pouco estudada [Sou97]. Gogolla e Richters [GR98] propõem a transformação de diagramas de classes UML que contenham restrições de cardinalidade, qualificadores, classes de associação, agregações, composições e generalizações, em diagramas UML que utilizem apenas associações n-árias e restrições OCL. Ferramentas comerciais, tais como a Rational Rose e Borland Together, não suportam a modelagem, muito menos a recuperação de associações n-árias, conforme levantamento feito por Kollmann e outros [KSS+02]. Em outro estudo, eles propõem a identificação de associações n-árias por meio da descoberta de classes que se associem a três ou mais classes, tendo multiplicidades '1' nas extremidades-alvo das associações que partem dessa classe central [KG01a]. Guéhéneuc [Gué04c] discorda dessa abordagem, "pois as especificações UML não restringem as multiplicidades de classificadores que participem em associações n-árias".
4. Lacuna semântica (semantic gap) entre linguagens de modelagem e de implementação. Guéhéneuc e outros [GAA04] afirmam que "linguagens de modelagem possuem alguns conceitos similares aos suportados por linguagens de programação, tais como classes e generalização, fornecendo alguma continuidade entre o desenho e sua implementação". Citando Guéhéneuc em seu levantamento dos elementos de diagramas de classes da UML [Gué04c], "a generalização é uma construção sintática presente na maioria das linguagens de programação orientadas por objeto; entretanto, o suporte semântico oferecido por cada uma dessas linguagens não é homogêneo: por exemplo, em relação à amarração dinâmica, co-variância e contra-variância". Para sanar esse problema, o autor sugere que generalizações, quando modeladas, deveriam ser anotadas com estereótipos ou anotações sobre a linguagem de programação a ser utilizada

ou das escolhas de implementação. Contudo, a modelagem atua no domínio do problema, lidando com conceitos que não são (ainda) suportados por linguagens de programação, tais como: associações simples, composições e agregações em diagramas de classe [GAA04]. Isso apresenta dificuldades para a recuperação desses relacionamentos:

- Segundo Kollmann e outros [KSS+02], como não há um mapeamento "um-para-um" entre os elementos dos diagramas de classe da UML e o código-fonte, composições e agregações podem ser implementadas de modo semelhante, embora sejam conceitualmente diferentes. "Associações, em geral, são de difícil detecção, principalmente em linguagens dinamicamente tipadas". Os autores afirmam que o reconhecimento de composições exige técnicas de análise dinâmica, pois esse relacionamento define restrições sobre as instâncias das classes e não sobre as classes [KSS+02]. Entretanto, em trabalho recente sobre identificação de composições, Milanova [Mil05] propõe o uso de inferência estática de posse (static inference ownership), com a qual obteve quase 100% de revocação em seus experimentos.
- Associações de agregação. Fowler [Fow05] afirma que "a agregação é estritamente sem significado, sendo uma das fontes de confusão mais freqüentes na UML". Rumbaugh a considera "um placebo da modelagem"[RJB05]!
- Briand e outros defendem que a distinção entre os tipos de associações exige análise semântica além da análise estática do código, pois composições implicam dependências entre os tempos de vida das classes componentes e a classe composta. A identificação de multiplicidades '*' em extremidades de associações pode demandar a investigação do uso de classes de coleção (por exemplo, `java.util.Hashtable`) no código [BLM03].
- Guéhéneuc [Gué04c] afirma que, por meio de análise estática, multiplicidades podem ser recuperadas investigando-se o código-fonte em busca de declarações de campos: se o campo for um arranjo ou uma coleção, pode-se deduzir que a multiplicidade seja '0..*'; caso contrário, '0..1'. Aprofundando essa análise, a inferência de criações de instâncias pode precisar melhor as multiplicidades, mas tais análises geralmente são indecidíveis. Dinamicamente, pode-se contar o número de instâncias criadas (similarmente, em seu trabalho sobre recuperação de associações n-árias em bancos de dados relacionais, Soutou [Sou97] conclui que a determinação de multiplicidades necessita da análise de instâncias de dados, i.e., execução de análises dinâmicas).
- Ainda segundo Guéhéneuc, dependências não podem ser recuperadas a partir de análise estática, porque sua recuperação depende da semântica planejada pelos desenvolvedores.

5. Dificuldades na recuperação por meio de bytecodes. Cooper e outros [CKvKR04], durante implementação da ferramenta REV-SA, identificaram informações que não puderam ser recuperadas do bytecode das classes Java, algumas das quais seguem abaixo:

   - operações que não alterem o estado do objeto (restrição `query`). Keschenau [Kes04] afirma detectar esse tipo de operações, mas não apresenta sua abordagem por limitações de espaço do artigo;
   - concorrência das operações (`sequential` ou `concurrent`);
   - distinção da direção dos parâmetros das operações (`in`, `out`, `in-out` ou `return`);

- distinção entre os valores iniciais: zero, `null`, `false`, Unicode 0000 e ausência de valor inicial;
- a extremidade de uma associação pode ser desordenada ou ordenada (`ordered`);
- a extremidade de uma associação pode permitir ou não duplicatas (`bag` e `set`, respectivamente);
- recuperação de nomes textuais de generalizações e dependências de abstração («`trace`», «`refine`» ou «`derive`»);
- herança múltipla.

Além disso, identificaram o acréscimo de elementos ao código das classes, feito pelo compilador Java, que afetam o diagrama de classes recuperado e o tornam inconsistente em relação a um mesmo diagrama gerado por engenharia direta. Por exemplo, generalizações para `java.lang.Object` e a criação de construtores-padrão (que invocam o construtor da super-classe) em classes que não o explicitam.

6. Dificuldades em análises dinâmicas. Kollmann e Gogolla [KG01a] afirmam que programas podem conter informações que não devem ser extraídas apenas do código-fonte, apresentando estruturas e comportamentos diferentes a cada execução. Isso pode se dever a diferentes entradas externas (parametrizações ou dados lidos de uma base de dados). Nesses casos, por exemplo, uma entrada externa pode determinar diferentes estratégias e tempos de vida dos objetos do programa, interferindo na detecção de agregações e composições. Dessa forma, informações recuperadas por meio de análises dinâmicas não têm sua validade assegurada em todos os casos.

4.14 Trabalhos relacionados

Outros trabalhos fizeram levantamento do estado-da-arte de engenharia reversa de sistemas Java para UML. Kollmann e outros [KSS+02] analisaram o desempenho de diferentes ferramentas na recuperação de elementos UML, segundo as seguintes propriedades do modelo: número de classes, de associações, seus tipos, manipulação de interfaces e classes de coleção, multiplicidades, nomes de papel, classes internas e detalhes de compartimento das classes (por exemplo, atributos, seu tipo e visibilidade). Por sua vez, Guéhéneuc [Gué04c] verificou de que maneira elementos de diagramas de classe da UML eram recuperados pelas diferentes ferramentas, segundo definições de "precisão" e "abstração" apresentadas pelos autores. Por fim, Arcelli e outros [AMRT05] estudaram ferramentas de engenharia reversa que utilizam o reconhecimento de sub-mecanismos de desenho, analisando vantagens e desvantagens de cada abordagem.

4.15 Iniciativas

Conforme discutido, não encontramos nenhuma abordagem que prescrevesse o uso de OCL no processo de engenharia reversa. Entretanto, acreditamos que usar OCL no resultado possa ser útil a engenheiros de software, permitindo variar a representação dos diagramas recuperados:

1. menos abstratamente, pode-se preservar informações em OCL, convertidas do modelo de implementação em Java;

2. ou tornar a interpretação mais abstrata, utilizando OCL para representar apenas os aspectos que se deseja mostrar.

Conforme o propósito a que se destinem os diagramas de classe recuperados, uma ou outra abordagem é mais desejável. Caso os diagramas sejam necessários para redocumentar softwares legados, mantenedores podem achar útil a primeira representação. Caso desenhistas de software desejem verificar se um dado modelo de implementação reflete o modelo de desenho respectivo, a segunda representação, mais abstrata, provavelmente será sua escolha.

A seguir, discutimos algumas de nossas iniciativas de utilização da OCL no processo de engenharia reversa. São apresentados: sua motivação, os estudos realizados e/ou tarefas executadas, e os problemas que inviabilizaram essa iniciativa.

Documentação de contratos de classes e interfaces de coleção. Qualquer ferramenta de engenharia reversa de Java precisará tratar da inferência de associações entre duas classes quando uma delas é uma classe de coleção. Ortogonais aos algoritmos utilizados, as restrições possuídas por essas classes e interfaces já se encontram documentadas no próprio código-fonte, na forma de comentários em *Javadoc*. Porém, essas restrições encontram-se descritas em linguagem natural, carecendo de uma maior precisão. O objetivo dessa iniciativa foi a conversão dessas restrições para sua forma em OCL, para que pudessem ser usadas pelas diferentes ferramentas de engenharia reversa na documentação dos diagramas gerados.

Realizou-se o levantamento das interfaces e classes do Arcabouço de Coleções Java. Partindo-se da interface mais genérica, `java.util.Collection`, os contratos de suas operações foram identificados (convertendo-os para OCL) e procuraram-se as restrições apresentadas por suas sub-interfaces, recursivamente, repetindo o processo. Por exemplo, seja a operação `boolean add(E o)`, da interface `Collection`. Essa operação adiciona um elemento à coleção, "i. garantindo que ela contenha o elemento especificado ao final da execução da operação e ii. retornando `true` se esta coleção mudou como um resultado da chamada.". Uma possível conversão dessas duas restrições, de linguagem natural para OCL, seria:

```
context Collection::add(E o): Boolean
   post elementoAdicionado: result = self->includes(o) and
      self@pre <> self
```

Ao fim da execução da operação (`post`), seu resultado (`result`) será a conjunção das condições: i. ao final da operação, a coleção contém o argumento (`self->includes(o)`); ii. a coleção, em seu estado anterior à execução da operação, é diferente da mesma coleção em seu estado posterior (`self@pre <> self`). Esse procedimento de conversão manual seria realizado para todas as operações encontradas. A partir de `Collection`, seriam percorridas todas as sub-interfaces, até que se atingissem as folhas da hierarquia de interfaces do arcabouço. Exemplificando, para a interface `java.util.Set` (abstração do *conjunto* matemático), a operação `add` possui as seguintes restrições adicionais: "(A operação) adiciona o elemento especificado a este conjunto se ele já não estiver presente. (...) Se este conjunto já contiver o elemento especificado, a chamada não altera o conjunto, retornando `false`.". Em OCL, tem-se:

```
context Set::add(E o): Boolean
```

```
post elementoAdicionadoSeAusente:
  result = self@pre->excludes(o) and self->includes(o)
```

Similar à operação anterior, ao fim da execução da operação, seu resultado será a conjunção das condições: i. a coleção, em seu estado anterior à execução da operação, não contém o argumento (self@pre->excludes(o)); ii. ao final da operação, a coleção contém o argumento (`self->includes(o)`). Bastaria, então, documentar os contratos especificados pelas interfaces que as classes implementam. Entretanto, algumas das operações das interfaces implementadas pelas classes desse arcabouço não são suportadas, disparando uma exceção em tempo de execução quando executadas (tipicamente: `ClassCastException`, `IllegalArgumentException` ou `NullPointerException`). Nesse caso, fez-se necessário identificar essas operações, alterando o contrato respectivamente.

Por fim, algumas das implementações de coleção possuem restrições quanto aos elementos que podem conter ou quanto aos tipos desses elementos. Por exemplo, a classe `java.util.EnumSet` não permite a inserção de elementos `null`. Essa restrição poderia ser escrita em OCL da seguinte forma (supondo o valor `OclUndefined`, de OCL, mapeado para o valor `null`, em Java):

```
context EnumSet::add(E o): Boolean
    pre naoInsereNulo: not ( o.oclIsUndefined() )
```

Antes da execução da operação (`pre`), é verificado se o argumento não é nulo (`not ( o.oclIsUndefined() )`)[2]

Julgamos que a conversão manual dos contratos presentes nos comentários em Javadoc para OCL não traria grandes benefícios. A cada nova versão de Java, seria necessário identificar manualmente as alterações feitas nas operações existentes das classes e interfaces de coleção, bem como as novas operações. A implementação e/ou re-utilização de um processador de linguagem natural para auxiliar na automatização do processo não seria viável, pois este não é o foco deste trabalho. Um problema adicional é que OCL não é capaz de expressar restrições de acesso concorrente e sincronização, tais como aquelas apresentadas por todas as implementações de propósito geral (por exemplo, `HashSet`, `ArrayList`).

Mudanças em campos de classe anotadas como pós-condições de operações. Essa iniciativa buscou identificar alterações realizadas em campos de uma classe, preservando essas transformações durante a engenharia reversa por meio de sua conversão para pós-condições em OCL.

Uma heurística simples, utilizada para a identificação de métodos que alterem o estado de um objeto, é a procura por aqueles cujo valor de retorno seja `void`. Todo o acesso de escrita dos campos de classe deveria ser analisado, realizando-se sua conversão para OCL. Por exemplo, seja a classe Java, *classeA*, abaixo:

```
class classeA {
```

[2] A operação `OclVoid::oclIsUndefined():Boolean` retorna `true` caso o objeto que a chamou seja igual a `OclUndefined`.

```
    private int a = 0;
    public void incrementaA() { this.a++; }
}
```

Convertida para OCL, a operação `incrementaA` poderia ser apresentada da seguinte maneira:

```
context classeA::incrementaA()
    post: self.a = self.a@pre + 1
```

Embora julguemos relevante a conversão das alterações de estado de um objeto após a execução de um método, casos reais são muito mais complexos que o exemplo simples mostrado acima. Alterações dos campos dentro de laços ou em operações recursivas necessitariam de análises estáticas sobre o código-fonte e/ou executável para inferir essas alterações (por exemplo, desenrolando os laços). O estudo dessas técnicas se encontra fora do escopo desse trabalho. Outra heurística que poderia ser usada é a suposição de que atribuições aos campos ocorra uma única vez, onde valores intermediários sejam calculados e armazenados em variáveis temporárias dos métodos. Entretanto, ainda seria necessário inferir o resultado dos campos caso estejam internos ao escopo de comandos condicionais em Java (por exemplo, `if`, `switch-case`), além de considerar a ocorrência de comandos de saída dos métodos ou do programa, como `return()` ou `System.exit()`.

Conversão de métodos de consulta Java para OCL. Quando avaliadas, expressões OCL não produzem efeitos colaterais. Dessa forma, sua avaliação não altera o estado do sistema sendo avaliado. Com base nisso, pensou-se na possibilidade de se criar um conversor de métodos Java que não alterem o estado do objeto, i.e. métodos de consulta, para OCL.

O algoritmo do método seria convertido para uma expressão OCL, utilizando a palavra-chave `body`. Para exemplificar, seja o seguinte método `ehMaiorDeIdade()`, em Java, que retorna `true` se a `idade` de uma `Pessoa` for maior ou igual a 21 anos:

```
class Pessoa {
  private int idade = 0;
  public boolean ehMaiorDeIdade() {
    return this.idade >= 21;
  }
}
```

Em OCL, ela poderia ser convertida para:

```
context Pessoa::ehMaiorDeIdade(): Boolean
    body: self.idade >= 21
```

Utilizando a palavra-chave `body`, é especificado como deve ser feito o cálculo da operação de consulta `ehMaiorDeIdade()`. Similar à iniciativa anterior, também é necessário o uso de técnicas de análise de código para desenrolar eventuais laços existentes. Além disso, toda a operação deveria ser convertida em uma única expressão, que seria o próprio valor retornado pela operação (requisito de `body` [OCL03]).

Documentação de exceções. Com essa iniciativa, tentou-se o uso de OCL para auxiliar a engenharia reversa e a frente de código Java, na documentação de invariantes, pré e pós-condições como exceções. Dessa forma, essas restrições gerariam automaticamente as exceções respectivas, que poderiam ser recuperadas utilizando engenharia reversa. A idéia foi propor uma forma pré-definida para tratamento de exceções em Java e UML.

UML 2.0 já suporta a documentação de exceções sem a necessidade de OCL. Além disso, Gurunath [Gur04] já havia proposto a documentação de exceções, utilizando comentários no modelo de desenho e convertendo-os para a restrição correspondente em Java. Antes dele, Soundarajan e Fridella [SF99] propuseram uma extensão de OCL para modelar exceções diretamente nesta linguagem.

Capítulo 5

Conclusão

Neste trabalho, nossa principal contribuição é apresentar o uso de OCL na engenharia direta de diagramas de classe UML para código Java e no possível uso de OCL no processo de engenharia reversa de código Java para diagramas de classe UML. Foi feito um levantamento do estado-da-arte de engenharia direta e de engenharia reversa, entre diagramas de classe de modelos de desenho UML e código-fonte Java. No contexto de engenharia direta, diferentes abordagens foram encontradas, apresentando-se os mapeamentos identificados entre os tipos, operações e operadores de OCL e de Java. No contexto de engenharia reversa, não foram encontradas ferramentas que suportassem o uso de OCL durante o processo, sendo analisadas aquelas que fazem a engenharia reversa a partir de Java para diagramas de classe de um modelo de desenho UML. Foram mostradas as diferentes formas de recuperação empregadas por essas ferramentas na identificação de relacionamentos de generalização, realização, associação simples, associação com adornos (agregação ou composição) e dependências. Tanto em engenharia direta quanto em engenharia reversa, realizaram-se comparações entre as ferramentas existentes, assim como foram levantados problemas identificados por outros autores durante a execução de cada processo. Ambicionávamos a implementação de uma ferramenta relacionada ao contexto do estado-da-arte do uso de OCL na engenharia direta de UML para Java, assim como na engenharia reversa de Java para UML. Entretanto, diferentes obstáculos se apresentaram, impedindo essa implementação. Nossas iniciativas nessa direção e as dificuldades enfrentadas foram também mostradas.

Baseando-se nos estudos realizados neste trabalho, apresentamos algumas conclusões sobre o uso de OCL durante o processo de desenvolvimento de software:

OCL ainda não é suficientemente madura para uso na indústria. A linguagem OCL possui uma grande capacidade de expressão, mas ainda não parece estar madura o suficiente para ser usada em sua totalidade. A especificação da linguagem apresenta uma série de ambigüidades e inconsistências [AP04, HZ04, ABB+05]. Ao longo da especificação, são encontradas chamadas a operações que não estão definidas nem referenciadas em lugar algum, bem como expressões incorretas nos próprios exemplos de uso [AP04, EmP05]. O significado de múltiplas pré-condições e pós-condições no contexto de uma operação não é explicitado [ABB+05]. Além disso, a linguagem permite a criação de expressões cuja computação necessária para avaliá-las pode ser muito cara (por exemplo: `self.b.c@pre`) [DBL05]. Todos esses problemas

podem gerar diferentes interpretações e/ou decisões de implementação. Isso parece indicar uma certa falta de entendimento sobre as definições dessas construções, bem como a existência de dificuldades que ainda não foram superadas pelos autores da especificação da linguagem. Para maiores detalhes, vide Seção 3.11.

OCL é difícil de compreender. A curva de aprendizado da linguagem não é suave, confundindo desenvolvedores iniciantes. Mesmo usuários experientes enfrentam dificuldades na interpretação de expressões complexas (para um exemplo não muito simples, vide a especificação da operação `rent`, na Figura 3 do trabalho de Correa e Werner [CW04] ou as regras de boa formação na especificação da UML 2.0 [UML05]). De acordo com Siikarla e outros [SPS04], embora expressões curtas e diretas sejam, freqüentemente, muito simples, sua clareza e legibilidade diminuem muito quando a complexidade e o tamanho dessas expressões aumentam. Além disso, apresentam-se muitas dificuldades ao se tentar conectar dois sistemas que possuam bases conceituais muito diferentes. Pode ser problemático exigir que um desenvolvedor saiba se expressar convenientemente em linguagens pertencentes a dois paradigmas distintos: Java, no paradigma imperativo; OCL, no declarativo. Por exemplo, na área de bancos de dados, isso ficou conhecido como o *problema de falha no casamento de impedância* (impedance mismatch problem)[1]. Além disso, os problemas existentes na especificação contribuem para dificultar o entendimento de OCL.

Processos de desenvolvimento de software ignoram a existência de OCL. Processos de desenvolvimento de software, tais como o Processo Unificado, o *EUP* (Enterprise Unified Process)[2], Praxis e o método *OPEN* (Object-oriented Process, Environment and Notation)[3] prescrevem o uso da linguagem UML na construção de modelos. Uma grande parte dos elementos da UML é utilizada por esses processos, sendo a forma de uso de alguns desses elementos explicitamente prescrita. Entretanto, não foi encontrada nesses processos qualquer prescrição do uso de expressões OCL durante o desenvolvimento. Dessa forma, permanece indefinido quem deve adicionar essas restrições aos diagramas UML (clientes, analistas ou desenhistas de software), quem deveria consumir essa informação, nem tampouco quando essa adição de restrições deve ocorrer durante o desenvolvimento. Entendemos que os autores de processos ainda não prescrevem o uso de OCL, entre outras coisas, por que não existem boas ferramentas que a suportem (essa falta de ferramentas também é apontada como uma das possíveis causas, por Briand e outros [BDL04]). Entretanto, é possível que ainda não existam boas ferramentas que suportem OCL porque os autores de processos não prescrevem seu uso, criando um círculo vicioso. Briand e outros [BDL04] especulam que haja preconceitos estabelecidos, entre especialistas em desenvolvimento de software e muitos metodologistas influentes, contra o uso de quaisquer elementos formais durante o desenvolvimento. Some-se a tudo isso os problemas na especificação da linguagem e sua complexidade, tem-se uma possível hipótese de por que os autores de processos ainda não prescrevem o uso de OCL.

[1] Isso ocorre quando um banco de dados relacional é usado por um programa escrito em uma linguagem orientada por objetos; particularmente, quando objetos e/ou definições de classe são mapeadas diretamente para tabelas de bancos de dados e/ou esquemas relacionais (relational schemata).

[2] http://www.enterpriseunifiedprocess.com

[3] http://www.open.org.au

Modelagem versus Implementação. Quanto maior a lacuna semântica entre as linguagens de modelagem e de implementação, mais difícil é a transição entre esses níveis. No sentido de aproximar a linguagem de implementação da linguagem de modelagem, é desejável que conceitos presentes no nível de modelagem sejam suportados em algum grau pela linguagem de programação onde serão implementados. Para exemplificar, alguns dos conceitos do paradigma de Projeto por Contrato suportados pela OCL – invariantes, pré-condições e pós-condições – carecem de suporte nativo da linguagem de implementação a ser usada, para que a transição de modelos de desenho para código-fonte seja menos problemática. Em Java, esses conceitos não se encontram modelados, mas há linguagens que os suportam, como *Eiffel*. Ela utiliza as palavras-chave `invariant`, `requires` e `ensures`, referindo-se, respectivamente, aos invariantes, pré-condições e pós-condições de OCL (mesmo o operador `@pre` de OCL é suportado em Eiffel, por meio do operador `old`).

No sentido oposto, para aproximar a linguagem de modelagem da linguagem de implementação, alguns autores sugerem o uso de JML. Segundo Burdy e outros [BCC+05], a *JML* (Java Modeling Language) é mais apropriada para especificação de contratos que OCL, pois a notação JML utiliza a sintaxe Java, tornando-a mais fácil de aprender para os programadores desta linguagem. Além disso, a semântica de JML é construída sobre a semântica de Java. Por exemplo, a operação `equals` e o operador "`==`" possuem o mesmo significado em JML e em Java, enquanto que, em OCL, é utilizado o operador "`=`". Os autores afirmam que, durante o desenvolvimento de uma aplicação Java a partir de modelos UML, "pode ser melhor utilizar JML ao invés de OCL para a especificação de restrições em objetos, especialmente nos estágios finais do desenvolvimento".

Dificuldades de recuperação de contratos na implementação. Segundo Kollmann e Gogolla [KG01a], utilizando engenharia direta, restrições e proibições em OCL presentes nos diagramas de classes devem ser implementadas no código gerado, estando "diluídas" nele. Entretanto, por meio de engenharia reversa, nem sempre é possível identificar exemplos que permitam a dedução dessas restrições. Informações de tempo de execução do software podem indicar certas restrições que sejam válidas para ele. Mas para provar que essas restrições não se sustentam, deve-se encontrar um exemplo negativo para cada. Nem sempre é possível determinar após quantas execuções ou sob que condições um exemplo negativo ocorrerá. Talvez a prescrição do uso de OCL por um processo de desenvolvimento de software possa fornecer subsídios para engenharia reversa de contratos. Além disso, parece ser necessário, a princípio, restringir o escopo de uma possível utilização de OCL no processo de engenharia reversa.

Os benefícios obtidos com a descrição mais precisa de modelos usando OCL podem compensar o esforço necessário para se treinar engenheiros de software nessa linguagem. Entretanto, isso deve ser ainda verificado. Como trabalhos futuros, seguem alguns exemplos de investigação:

1. *Estudo experimental.* A adição de uma maior formalização ao processo de desenvolvimento proporciona um aumento da qualidade do software sendo produzido. Utilizando OCL para descrever, com maior precisão, modelos da UML, queremos montar um experimento que descubra se o aumento da qualidade do produto construído compensa o aumento de custo que certamente será incorrido.

2. *Investigação de base histórica de defeitos.* Com essa abordagem, almejamos descobrir se é possível obter economia de recursos de um projeto de desenvolvimento de software (esforço e tempo) com a descrição mais precisa de modelos, utilizando OCL. Serão levantados os defeitos ocorridos em projetos de uma organização de desenvolvimento de software, a partir de sua base histórica de projetos. O objetivo é identificar defeitos que poderiam ser evitados (ou pelo menos, minimizada a possibilidade de ocorrência), caso os modelos utilizassem restrições OCL. O tempo e esforço que foram necessários para a remoção desses defeitos serão confrontados com o tempo e esforço necessários ao uso de OCL no projeto.

Referências Bibliográficas

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool - integrating object oriented design and formal verification. *Software and System Modeling*, 4(1):32–54, 2005.

[AL03] David Akehurst and Peter Linington. OCL 2.0: Implementing the standard. Technical Report TR-12-03, Computing Laboratory, University of Kent, Canterbury, UK, 2003.

[AMRT05] Francesca Arcelli, Stefano Masiero, Claudia Raibulet, and Francesco Tisato. A comparison of reverse engineering tools based on design pattern decomposition. In *Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)*, 2005.

[AP04] David H. Akehurst and Octavian Patrascoiu. OCL 2.0 - implementing the standard for multiple metamodels. *Electronic Notes in Theoretical Computer Science*, 102:21–41, 2004.

[APM03] G. Antoniol, M. Di Penta, and E. Merlo. YAAB - Yet Another AST Browser - Using OCL to Navigate ASTs. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension.*, pages 13 – 22, May 2003.

[arg05] ARGO/UML project, ARGO/UML project home page, 2005. Disponível em: http://argouml.tigris.org/.

[Baa05] Thomas Baar. Non-deterministic constructs in OCL - what does any() mean. In Springer, editor, *12th SDL Forum*, volume 3530, pages 32–46, Grimstad, Norway, 2005.

[BAMF04] D. Barbosa, W. Andrade, P. Machado, and J. Figueiredo. SPACES - uma ferramenta para teste funcional de componentes. In *XI Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software (SBES'04)*, pages 55–60, Brasília, Brazil, Oct 2004.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212 – 232, June 2005.

[BDL04] Lionel C. Briand, W. Dzidek, and Y. Labiche. Using aspect-oriented programming to instrument OCL contracts in Java. Technical Report SCE-04-03, Software Quality Laboratory, Carleton University, February 2004.

[BGG04] Hanna Bauerdick, Martin Gogolla, and Fabian Gutsche. Detecting OCL Traps in the UML 2.0 Superstructure: An Experience Report. In Baar et al. [BSMM04], pages 188–196.

[BLM03] L. C. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 57 – 66, November 2003.

[BLS03] L.C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Software - Practice and Experience*, 33(7):637–672, 2003.

[BRC06] Artur Boronat, Isidro Ramos, and José Á. Carsí. Definition of OCL 2.0 operational semantics by means of a parameterized algebraic specification. In *1st International Workshop on Algebraic Foundations for OCL and Applications*, Valencia, Spain, March 2006.

[BSMM04] Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors. *UML 2004 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings*, volume 3273 of *Lecture Notes in Computer Science*. Springer, 2004.

[BW05] Lionel Briand and Clay Williams, editors. *MODELS / UML 2005 - The Unified Modelling Language: Modelling Languages and Applications. ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, October 2-7, 2005. Proceedings*, volume 3713 of *Lecture Notes in Computer Science*. Springer, 2005.

[Bèz05] Jean Bèzivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[CKvKR04] David Cooper, Benjamin Khoo, Brian R. von Konsky, and Michael Robey. Java implementation verification using reverse engineering. In *Proceedings of the 27th Conference on Australasian Computer Science*, pages 203 – 211, Dunedin, New Zealand, 2004.

[CL01] S. Chung and Y. S. Lee. Reverse software engineering with UML for web site maintenance. In *Proceedings of the 1st International Conference on Web Information Systems Engineering (WISE'00)*, volume 2, pages 157 – 161, Hong-Kong, China, June 2001.

[CW04] Alexandre Correa and Cláudia Werner. Applying Refactoring Techniques to UML/OCL. In Baar et al. [BSMM04], pages 173–187.

[DBL05] Wojciech Dzidek, Lionel Briand, and Yvan Labiche. Lessons learned from developing a dynamic OCL constraint enforcement tool for Java. In Briand and Williams [BW05], pages 10–19.

[DS05] Frank Devos and Eric Steegmans. Specifying business rules in object-oriented analysis. *Software and System Modeling*, 4:297–309, 2005.

[EmP05] EmPowerTec. Analysis of the expressions from the OCL 2.0 specification, Oct 2005. Disponível em http://www.empowertec.de/products/analyze-spec-expressions.htm.

[Fil03] Wilson Filho. *Engenharia de Software - Fundamentos, Métodos e Padrões*. LTC, 2nd edition, 2003.

[Fow05] Martin Fowler. *UML Essencial - Um breve guia para a linguagem-padrão de modelagem de objetos*. Bookman, Porto Alegre, 3rd edition, 2005.

[GAA04] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships - putting icing on the UML cake. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 301 – 314, Vancouver, BC, Canada, 2004.

[GBR03] Martin Gogolla, Jörn Bohling, and Mark Richters. Validation of UML and OCL Models by Automatic Snapshot Generation. In Grady Booch, Perdita Stevens, and Jonathan Whittle, editors, *Proceedings of the 6th International Conference on Unified Modeling Language (UML'2003)*, pages 265–279, San Francisco, CA, USA, 2003. Springer, Berlin, LNCS 2863.

[GBR05] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[GDJ02] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – a tool for dynamic analysis of Java programs. In Wolfgang Emmerich and Dave Wile, editors, *Proceedings of the 17th Conference on Automated Software Engineering*, pages 117 – 126. IEEE Computer Society Press, September 2002.

[GH04] Martin Giese and Rogardt Heldal. From Informal to Formal Specifications in UML. In Baar et al. [BSMM04], pages 197–211.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison Wesley, 27th edition, 1994.

[GHL04] Martin Giese, Reiner Hähnle, and Daniel Larsson. Rule-Based Simplification of OCL Constraints. In Baar et al. [BSMM04].

[GL05] Martin Giese and Daniel Larsson. Simplifying transformations of OCL constraints. In Briand and Williams [BW05], pages 309–323.

[GR98] Martin Gogolla and Mark Richters. Equivalence rules for UML class diagrams. Technical report, University of Bremen, 1998.

[GS04] Ahmed Gaafar and Sherif Sakr. Towards a framework for mapping between UML/OCL and XML/XQuery. In Baar et al. [BSMM04], pages 241–259.

[GSMD03] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards Automating Source-Consistent UML Refactorings. In Grady Booch, Perdita Stevens, and Jonathan Whittle, editors, *Proceedings of the 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML'2003)*, pages 144–158, San Francisco, CA, USA, October 2003. Springer, Berlin, LNCS 2863.

[GSZ04] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In Eleni Stroulia and Andrea de Lucia, editors, *Proceedings of the* 11^{th} *Working Conference on Reverse Engineering*, pages 172–181. IEEE Computer Society Press, November 2004.

[Gué04a] Yann-Gaël Guéhéneuc. Abstract and precise recovery of UML class diagram constituents. In *Proceedings of the 20th IEEE International Conference on Software Maintenance, 2004*, page 523, September 2004.

[Gué04b] Yann-Gaël Guéhéneuc. A reverse engineering tool for precise class diagrams. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 28 – 41, Markham, Ontario, Canada, 2004.

[Gué04c] Yann-Gaël Guéhéneuc. A systematic study of UML class diagram constituents for their abstract and precise recovery. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 265 – 274, December 2004.

[Gué05] Yann-Gaël Guéhéneuc. Ptidej: Promoting patterns with patterns. In *Proceedings of the* 1^{st} *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005.

[Gur04] Pramod Gurunath. OCL exception handling. Master's thesis, Texas A&M University, August 2004.

[HDF00] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In *Proceedings of the 3th International Conference on Unified Modeling Language (UML'2000)*, pages 51–69, York, UK, October 2000. Springer.

[HJR02] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An Authoring Tool for Informal and Formal Requirements Specifications. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering (FASE), Part of*

Joint European Conferences on Theory and Practice of Software, ETAPS, Grenoble, volume 2306 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2002.

[HS05] Brian Henderson-Sellers. UML- the Good, the Bad or the Ugly? Perspectives from a panel of experts. *Software and System Modeling*, 4(1):4–13, 2005.

[HZ04] Heinrich Hussmann and Steffen Zschaler. The Object Constraint Language for UML 2.0 – Overview and Assessment. *UPGRADE, Digital Journal of CEPIS (Council of European Professional Informatics Societies)*, V(2):25–28, April 2004.

[JBR98] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1st edition, 1998.

[JW99] Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. In David Garlan and Jeff Kramer, editors, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 194 – 202. ACM Press, May 1999.

[Kes04] Martin Keschenau. Reverse engineering of UML specifications from Java programs. In *Conference on Object Oriented Programming Systems Languages and Applications, Companion to the 19th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 326 – 327, Vancouver, BC, Canada, 2004.

[KG01a] R. Kollmann and M. Gogolla. Application of UML associations and their adornments in design recovery. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 81 – 90, October 2001.

[KG01b] R. Kollmann and M. Gogolla. Capturing dynamic program behavior with UML collaboration diagrams. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 58 – 67, March 2001.

[KG02] R. Kollmann and M. Gogolla. Metric-based selective representation of UML diagrams. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 89 – 98, March 2002.

[KSS+02] R. Kollmann, P. Selonen, E. Stroulia, T. Systa, and A. Zundorf. A study on the current state of the art in tool-supported UML-based static reverse engineering. In *Proceedings of the 9th Working Conference on Reverse Engineering*, pages 22 – 32, November 2002.

[Lad03] Ramnivas Laddad. *AspectJ in Action - Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[LB05] Luigi Lavazza and Giancarlo Barresi. Automated support for process-aware definition and execution of measurement plans. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 234 – 243, St. Louis, MO, USA, May 2005.

[LO03] Sten Loecher and Stefan Ocke. A metamodel-based OCL-compiler for UML and MOF. In Elsevier, editor, *Proceedings of the 6th International Conference on the Unified Modelling Language and its Applications (UML'03)*, volume 154, San Francisco, CA, USA, 2003.

[MB05] Slavisa Markovic and Thomas Baar. Refactoring OCL annotated UML class diagrams. In Briand and Williams [BW05], pages 280–294.

[MCL04] Jeffrey Mak, Clifford Choy, and Daniel Lun. Precise Modeling of Design Patterns in UML. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, Scotland, May 2004.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[MH69] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[Mil05] Ana Milanova. Precise identification of composition relationships for UML class diagrams. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (ASE '05)*, pages 76 – 85, Long Beach, California, USA, November 2005. ACM Press.

[MT04] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126 – 139, February 2004.

[NNWZ00] Ulrich A. Nickel, Jörg Niere, Jörg P. Wadsack, and Albert Zündorf. Roundtrip engineering with FUJABA. In J. Ebert, B. Kullbach, and F. Lehner, editors, *Proceedings of the 2^{nd} Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*. Fachberichte Informatik, Universität Koblenz-Landau, August 2000.

[NNZ00] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 742–745, Limerick, Ireland, 2000.

[OCL03] OMG UML 2.0 OCL specification, Oct 2003. Disponível em http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf.

[OK03] L. Ol'khovich and D. V. Koznov. OCL-Based Automated Validation Method for UML Specifications. *Programming and Computer Software*, 29(6):323–327, 2003.

[PE00] Magnus Penker and Hans-Erik Eriksson. *Business Modeling with UML: Business Patterns at Work*. John Wiley and Sons, 2000.

[RG03] Mark Richters and Martin Gogolla. Aspect-oriented monitoring of UML and OCL constraints. In Omar Aldawud, Mohamed Kande, Grady Booch, Bill Harrison, Dominik Stein, Jeff Gray, Siobhan Clarke, Aida Zakaria, Peri Tarr, and Faisal Akkawi, editors, *Proceedings of the UML 2003 Workshop Aspect-Oriented Software Development with UML*. Illinois Institute of Technology, Department of Computer Science, http://www.cs.iit.edu/∼oaldawud/AOM/index.htm, 2003.

[Ric02] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[RJB05] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2nd edition, 2005.

[SF99] Neelam Soudarajan and Stephen Fridella. Modeling exceptional behavior. In *Proceedings of the 6th International Conference of the Unified Modeling Language (UML'1999)*, 1999.

[SK04] Shane Sendall and Jochen Küster. Taming model round-trip engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 2004.

[Smi02] Jason McC. Smith. An elemental design patterns catalog. Technical Report 02-040, University of North Carolina, Chapel Hill, December 2002.

[Sou97] Christian Soutou. Knowledge discovery in relational database: extraction of n-ary relationships. In *Proceedings of the Third Basque International Workshop on Information Technology (BIWIT '97)*, pages 46 – 53, July 1997.

[SPS04] Mika Siikarla, Jari Peltonen, and Petri Selonen. Combining OCL and programming languages for UML model processing. *Electronic Notes in Theoretical Computer Science*, 102:175–194, 2004.

[SS03a] Jason McC. Smith and David Stotts. Elemental design patterns and the rho-calculus: Foundations for automated design pattern detection in SPQR. Technical Report 03-032, Computing Science Department, University of North Carolina, Chapel Hill, September 2003.

[SS03b] Jason McC. Smith and David Stotts. SPQR: flexible automated design pattern extraction from source code. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 215 – 224, October 2003.

[SvG98] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of Java software. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 10–16, Lake Buena Vista, Florida, United States, 1998. SIGSOFT (ACM Special Interest Group on Software Engineering) and

SIGPLAN (ACM Special Interest Group on Programming Languages), ACM Press - New York, NY, USA.

[UML03] OMG UML 1.5 specification, Mar 2003. Disponível em http://www.omg.org/docs/formal/03-03-01.pdf.

[UML04] OMG UML 2.0 infrastructure specification, November 2004. Disponível em http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-14.pdf.

[UML05] OMG UML 2.0 superstructure specification, August 2005. Disponível em http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf.

[Ver01] Bart Verheecke. From declarative constraints in conceptual models to explicit constraint classes in implementation models. Master's thesis, Vrije Universiteit Brussel, Faculteit van de Wetenschappen - Departement Informatica, May 2001.

[VPH+04] Amanda Varella, Vinícios Pereira, Vinícius Von Held, Geraldo Zimbrão, and João C. P. da Silva. Uma interface em linguagem natural para a verificação de regras de negócio em bases de dados. In *4o. Simpósio de Desenvolvimento e Manutenção de Software da Marinha (SDMS'04)*, 2004. http://www.mar.mil.br/sdms/programacao.htm.

[VS02] Bart Verheecke and Ragnhild Van Der Straeten. Specifying and implementing the operational use of constraints in object-oriented applications. In *Proceedings of the* 40^{th} *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 23–32, Sydney, Australia, 2002.

[Wan03] Mitchell Wand. Understanding aspects (extended abstract). In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*, volume 38, August 2003.

[WK03] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2nd edition, August 2003.

[ZA05] Uwe Zdun and Paris Avgeriou. Modeling architectural patterns using architectural primitives. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 133 – 146, San Diego, California, USA., October 2005.