PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EM DIREÇÃO A UMA SEMÂNTICA DA LINGUAGEM DE DESCRIÇÃO DE REUSO EM UML/OCL

THAIZE RORATO

Dissertação apresentada como requisito para obtenção do grau de Mestre, pelo Programa de Pós-Graduação da Faculdade de Informática da Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Dr. Toacy Cavalcante de Oliveira

Porto Alegre

Maio de 2007

Dados Internacionais de Catalogação na Publicação (CIP)

R787e Rorato, Thaize
Em direção a uma semântica da linguagem de descrição de reuso em UML/OCL / Thaize Rorato. – Porto Alegre, 2007.
84 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS
Orientador: Prof. Dr. Toacy Cavalcante de Oliveira

1. Orientação a Objetos. 2. Framework. 3. Programação (Computadores). 4. Linguagens de Programação. 5. Informática. 6. Engenharia de Software. I. Título.

CDD 005.114

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

PUCRS
Campus Central
Av. Ipiranga, 6681 - prédio 16 - CEP 90619-900
Porto Alegre - RS - Brasil
Fone: +55 (51) 3320-3544 - Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Em Direção a uma Semântica da Linguagem de Descrição de Reuso em UML/OCL***", apresentada por Thaize Rorato, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas de Informação, aprovada em 11/05/2007 pela Comissão Examinadora:

Prof. Dr. Toacy Cavalcante de Oliveira – Orientador — PPGCC/PUCRS

Prof. Dr. Marcelo Blois Ribeiro – PPGCC/PUCRS

Prof. Dr. Flávio Moreira de Oliveira – FACIN/PUCRS

Homologada em 26/11/07, conforme Ata No. 24 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

AGRADECIMENTOS

"Make it a habit to tell people thank you. To express your appreciation, sincerely and without the expectation of anything in return. Truly appreciate those around you, and you'll soon find many others around you. Truly appreciate life, and you'll find that you have more of it".
Ralph Marston

Meu maior agradecimento é dirigido aos meus pais por todo o apoio e esforço empenhado para que eu pudesse vir para Porto Alegre cursar o mestrado. Agradeço de coração a minha mãe querida, por todas as orações, força, paciência e sua crença absoluta na minha capacidade de realização deste trabalho. Agradeço ao meu pai por toda a força, conselhos e incentivos nas horas difíceis. Rafael, meu noivo, soube compreender a fase pela qual eu estava passando, durante a realização desta dissertação, sempre tentou entender minhas dificuldades e minhas angústias, sempre disposto a me ouvir e me acalmar. Agradeço-lhe, carinhosamente, por tudo.

Ao meu orientador Toacy, agradeço a consideração de ter aceitado a orientação de minha dissertação, obrigada por todos os ensinamentos e pela oportunidade creditada a mim.

A todos os professores, funcionários e alunos do Programa de Pós Graduação em Ciência da Computação da PUC-RS e todos aqueles que, direta ou indiretamente, contribuíram para a realização deste trabalho, me dando força e incentivo.

Agradeço a meus familiares, obrigada vó Nilza, tia Maíra, tia Márcia, tio Beto, Gabi, tia Simone, tio Zebrão, obrigada família!

A meus amigos Beto, Mariana, Gustavo, Karina, Elisa, Chris, Taísa e a todos que contribuíram com sua amizade, gostaria de expressar minha profunda gratidão.

Agradeço à Quantiza Systems pela bolsa concedida durante uma parte do curso.

Muito obrigada a todos!

Resumo

O processo de reutilização de frameworks é chamado de processo de instanciação. Para que a representação das atividades de instanciação esteja correta existe a necessidade de estender a formalização da RDL. A Linguagem de Descrição de Reuso – RDL – está formalizada através da sua BNF (Backus-Naur-Form). Esta formalização não define todas as restrições necessárias que devem ser aplicadas aos modelos construídos a partir da RDL. A Linguagem de Descrição de Reuso permite a representação das atividades de instanciação de frameworks orientados a objetos. O objetivo deste trabalho é estender a formalização da linguagem através da elaboração do seu meta-modelo utilizando um diagrama de classes UML; e através da descrição de um conjunto de restrições aplicáveis ao meta-modelo, escritas em linguagem natural e OCL (Linguagem de Restrição de Objetos), com o objetivo de detectar ambigüidades e inconsistências dos modelos.

Palavras Chave: Linguagem de Descrição de Reuso (RDL), Linguagem de Restrição de Objetos (OCL) e Well Formedness Rules (Regras de Boa Formação).

Abstract

The framework reuse process is called instantiation process. For the representation of the instantiation activities be considered correct it is necessary to extend the RDL formalization. The Reuse Description Language – RDL – is formalized through BNF (Backus-Naur-Form). This kind of formalization doesn´t defines all necessary constraints that should be applied to the models constructed with RDL. The RDL language allows the representation of the instantiation of object oriented framework activities. The aim of this work is to extend the RDL formalization through the definition of RDL metamodel using a class diagram; and through the description of a set of constraints applicable to the metamodel, wrote in natural language and OCL (Object Constraint Language), to detect ambiguity and inconsistency of the models.

Keywords: Reuse Description Language, Object Constraint Language and Well Formedness Rules.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

BNF	Backus-Naur-Form
CASE	Computer Aided Software Engineering
DSL	Domain Specific Language
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
RDL	Reuse Description Language
UML	Unified Modeling Language
UML-FI	Framework de Instanciação UML
XMI	Metadata Interchange

Sumário

1 INTRODUÇÃO

O aumento da complexidade dos sistemas de informação requer que os modelos construídos sejam corretos e, dependendo das características do sistema desejado, da forma menos ambígua possível [SOM05]. Quando se trata com sistemas complexos e de grande porte deve-se atentar para a qualidade dos serviços e produtos para evitar a ocorrência de problemas oriundos de sistemas mal projetados e mal desenvolvidos.

Desta forma, existe a necessidade de especificações precisas a fim de que se tenha uma documentação apropriada, que permita compreender, de forma mais fácil, as características essenciais do sistema, bem como as alternativas de processos que venham contribuir para a melhoria e eficácia dos sistemas. Para isso, são utilizadas linguagens que utilizam rigor matemático, de forma a possibilitar o uso de provas e testes de completude e consistência [BOW94a].

Deve-se ressaltar que existem limitações para se retratar certas propriedades do sistema, tendo em vista as características do sistema matemático escolhido, além do fato de serem estes métodos formais, em sua maioria, não atraentes a alguns usuários e desenvolvedores de sistemas. Esta dificuldade natural ocorre pela necessidade de uma base matemática mais aprofundada, para que se possa empregar e usufruir de toda a técnica formal [WAR99]. A motivação para o uso desta abordagem está na possibilidade de gerar automaticamente programas que sejam corretos por construção. Isso porque o próprio processo de desenvolvimento garantiria que o programa faz exatamente o que foi especificado.

O uso de soluções existentes no desenvolvimento de novos sistemas é um dos fatores chave para se alcançar melhorias na construção de sistemas. Reuso propõe o desenvolvimento de sistemas a partir de produtos de software pré-existentes, de modo que semelhanças entre requisitos e arquiteturas de diversos sistemas possam ser exploradas de uma forma melhor. A identificação destas semelhanças pode resultar, entre outras coisas, em significativas reduções de custo e tempo, isto é, aumento de produtividade para as empresas desenvolvedoras de software. Além disso, a cada uso do produto, sua eficiência e validade são verificadas, garantindo, assim, maior qualidade e redução de erros nos produtos finais [VAC01].

O reuso de código e de projeto pode ser feito a partir de frameworks orientados a objetos, que são estruturas de classes que constituem implementações incompletas de sistemas que, estendidas, permitem produzir diferentes artefatos de software. A grande vantagem desta abordagem é a promoção de reuso de código e projeto, que pode diminuir o tempo e o esforço exigidos na produção de software [RIC01]. Em contrapartida, é complexo desenvolver frameworks, bem como aprender a usá-los. Existe uma curva de aprendizado, que é o tempo necessário para se tornar apto para obter vantagens do processo de reutilização.

O processo de reutilização de frameworks é chamado de processo de instanciação. É durante este processo que os pontos de extensão existentes no framework são preenchidos para se obter a aplicação final [VAC01].

RDL é uma linguagem imperativa que suporta a representação das atividades de instanciação de um framework em um script chamado *Cookbook*, que pode ser processado por uma ferramenta, guiando os reutilizadores no processo de instanciação [OLI04]. As atividades de instanciação em RDL são relacionadas à manipulação de modelos de classes e seqüência, customização de pontos de extensão específicos por meio de atividades oriundas da programação orientada a objetos. A representação do projeto é caracterizada por Diagramas de Classe UML (Unified Modeling Language). Estes diagramas descrevem os tipos de objetos presentes no sistema e os vários tipos de relacionamentos estáticos entre eles, mostram as propriedades e operações de uma classe e as restrições que se aplicam à maneira como os objetos são conectados. Um aspecto importante da abordagem é a possibilidade de representar atividades de instanciação concorrentes, que facilitam a especificação do processo de instanciação distribuído, que são largamente utilizados no cenário de Desenvolvimento de Software Global para reduzir os custos de desenvolvimento [HER01].

A proposta desta pesquisa surge da premissa de que para que a especificação do processo de instanciação esteja precisa, ela deve estar consistente, completa e com o mínimo de ambigüidades possível; neste contexto é necessário estender a formalização da linguagem RDL. A formalização detecta a ocorrência de várias situações incorretas, entre elas a extensão de uma classe inexistente no modelo. Em RDL o modelo está descrito de forma textual, os desenvolvedores de frameworks podem expressar como a instanciação do framework deve ser executada, listando as tarefas de instanciação em um script. Os scripts RDL são instâncias do meta-modelo RDL. Atualmente todas as restrições aplicadas aos modelos construídos em RDL estão descritas no código do compilador, mas nem todas as restrições necessárias para a construção de um modelo consistente estão definidas. O propósito deste trabalho é definir o meta-modelo de RDL, baseando-se na sua sintaxe concreta, a BNF, uma notação utilizada para expressar a estrutura estática de uma linguagem, e a partir do meta-modelo definir as restrições que devem ser aplicadas aos modelos construídos a partir do meta-modelo.

A Linguagem de Descrição de Reuso manipula modelos UML. Para garantir que esta manipulação seja coerente é necessário utilizar técnicas formais como OCL e/ou well formedness rules (regras de boa formação) na definição de um conjunto de restrições para reduzir as possíveis inconsistências dos modelos gerados a partir do meta-modelo RDL.

A formalização de RDL está dividida em Formalização Estática e Formalização Dinâmica. Na Formalização Estática são definidas restrições estruturais dos modelos, como a criação de novos métodos e atributos. Na Formalização Dinâmica são avaliadas as restrições que devem ser verificadas em tempo de execução, como por exemplo, na criação de uma nova

classe, deve-se verificar se já existe alguma classe com o mesmo nome, pois não é permitida a existência de classes com o mesmo nome nos modelos.

Um exemplo relacionado à proposta de formalização da linguagem RDL é o uso de Action Semantics; elas são utilizadas para manipular elementos UML, como por exemplo, transformação de modelos. A proposta da Action Semantics é fornecer um meta-modelo integrado ao meta-modelo da UML e um modelo de execução. Action Semantics podem ser combinadas com regras escritas em OCL para verificar se uma transformação ou um conjunto de transformações no modelo podem ser aplicadas a um dado contexto [SUN01], compartilhando desta forma, do mesmo objetivo da definição do meta-modelo RDL e do uso de restrições em OCL propostos nesta dissertação.

Outra abordagem semelhante é a BasicMTL, uma linguagem para transformação de modelos dependente de meta-modelos. A linguagem manipula modelos oriundos de qualquer tipo de meta-modelo e em qualquer tipo de repositório. A linguagem possui uma biblioteca padrão que define tipos primitivos derivados dos tipos da OCL 2.0 [MTL05]. O uso da abordagem de meta-modelos e OCL na sintaxe abstrata da linguagem BasicMTL faz com que os modelos criados com a linguagem sejam mais corretos e confiáveis.

1.1 Objetivos

A UML possui um meta-modelo a partir do qual foi definido um conjunto de restrições escritas em linguagem natural e OCL com o objetivo de aumentar a legibilidade dos modelos criados a partir deste meta-modelo [UML99].

A Linguagem de Descrição de Reuso atualmente está formalizada através da sua BNF. A BNF é utilizada na especificação formal da sintaxe de linguagens de programação. O Anexo I apresenta a BNF da RDL.

O objetivo deste trabalho é estender a formalização da linguagem RDL para enriquecer a qualidade dos modelos gerados a partir da linguagem. Os objetivos específicos são: *(i)* a partir da BNF, derivar o meta-modelo de RDL, utilizando um diagrama de classes UML, integrado com o meta-modelo da UML. *(ii)* definir um conjunto de restrições em linguagem natural que incorporam propriedades e comportamentos presentes em RDL e em seguida mapear estas restrições para a linguagem de restrição de objetos (OCL). Estas restrições não estão definidas formalmente, de uma forma sistemática, elas serão derivadas a partir de relatórios técnicos e manuais criados pelos autores da linguagem. *(iii)* testar a abordagem utilizando um compilador OCL.

1.2 Metodologia de Desenvolvimento

A Figura 1 descreve a metodologia de desenvolvimento deste trabalho e a seqüência das atividades realizadas. A partir da BNF da linguagem RDL e do meta-modelo da linguagem UML foi realizada a primeira atividade: foi definido o meta-modelo RDL. Este meta-modelo foi descrito através de um diagrama de classes utilizando a ferramenta USE, caracterizando a segunda atividade.

Para um modelo RDL ser considerado sintaticamente correto, ele deve ser uma instância consistente do meta-modelo RDL. A consistência desta instância é garantida através da estrutura do meta-modelo, por exemplo, multiplicidades nas associações e um conjunto de regras de boa formação expressadas em OCL.

Inicialmente foram descritas em linguagem natural as restrições lógicas aplicáveis aos elementos dos modelos. Em seguida, na atividade 3, foram definidas na ferramenta USE as restrições OCL correspondentes. No processo de testes do uso das restrições, os scripts RDL com a descrição do processo de instanciação (atividades de programação orientada a objetos, como extensão de classes, redefinição de métodos) foram declarados de forma textual na ferramenta – atividade 4. Na atividade 5 é disponibilizada a avaliação do processo (avaliação da integridade do sistema descrito no script).

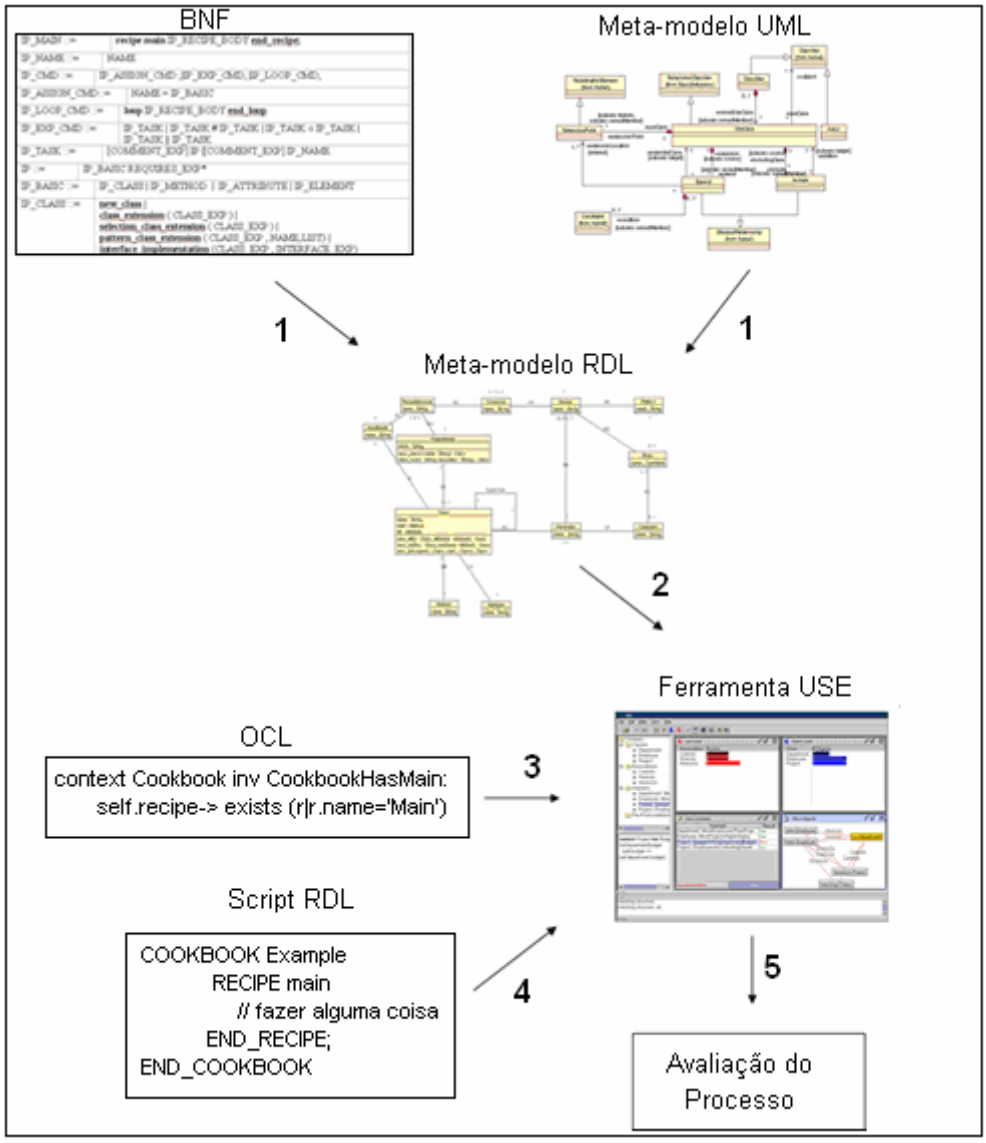

Figura 1 – Metodologia de Desenvolvimento.

1.3 Contribuições

As principais contribuições obtidas com a realização deste trabalho são:

- **Meta-modelo da linguagem RDL:** um diagrama de classes UML de acordo com a BNF definida pelos autores da linguagem.
- **Conjunto de restrições em linguagem natural:** aplicáveis aos comandos disponibilizados pela linguagem, estas restrições incorporam propriedades e comportamentos presentes na RDL.
- **Conjunto de restrições OCL:** restrições estáticas mapeadas para a linguagem OCL.
- **Exemplos de utilização:** foram utilizados scripts RDL para testar a aplicabilidade das

restrições definidas em OCL. Estes scripts foram executados na ferramenta USE, possibilitando a verificação da corretude do resultado das avaliações das restrições.

1.4 Organização do Documento

Este trabalho está organizado como segue. O Capítulo 2 apresenta uma fundamentação teórica para este trabalho, são introduzidos conceitos relacionados a métodos formais, é apresentada uma visão geral da Linguagem de Modelagem Unificada. Neste capítulo é dada ênfase ao meta-modelo da linguagem e ao uso de diagramas de classe. É apresentada a Linguagem OCL - Linguagem de Restrição de Objetos. Também são apresentados os conceitos de frameworks e o processo de instanciação de frameworks orientados a objetos.

O Capítulo 3 introduz a Linguagem de Descrição de Reuso, é apresentada uma visão geral da abordagem e os principais conceitos da linguagem.

O Capítulo 4 apresenta trabalhos relacionados com esta pesquisa. São apresentadas duas abordagens: Action Semantics e BasicMTL juntamente com seus principais conceitos.

O Capítulo 5 apresenta o conceito de formalização de linguagens. É apresentado o meta-modelo da RDL, a formalização estática e a formalização dinâmica, os comandos RDL e as restrições no uso destes comandos, definidas em linguagem natural, sendo esta a primeira contribuição deste trabalho.

O Capítulo 6 apresenta a ferramenta USE e as restrições definidas no capítulo 5 mapeadas em OCL utilizando a ferramenta USE para testar a corretude da implementação, sendo esta a segunda contribuição deste trabalho.

Por fim, as conclusões e os trabalhos futuros são apresentados no Capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é introduzido o conceito de métodos formais, é apresentada uma visão geral da Linguagem de Modelagem Unificada, é dada ênfase ao meta-modelo da linguagem e ao uso de diagramas de classe. A linguagem OCL (Object Constraint Language) é introduzida e são apresentados os conceitos de frameworks e o processo de instanciação de frameworks orientados a objetos.

2.1 Métodos Formais

A crescente exigência por sistemas confiáveis e de boa qualidade deu origem a muitas metodologias para o desenvolvimento de sistemas computacionais. Os métodos formais são muito utilizados atualmente na especificação de sistemas complexos com o objetivo de construir sistemas de forma mais sistemática e reduzindo ambigüidades.

Os métodos formais envolvem um conjunto de ferramentas e notações – com uma semântica formal – utilizadas para especificar de forma não ambígua os requisitos de um sistema [WIR98]. O uso de representações formais, ou semiformais, pode ser visto como uma abordagem para obter e verificar a consistência de modelos. Entretanto, especificações formais são complexas, em função do rigor matemático das linguagens, além de dificuldade de integração com ferramentas de desenvolvimento com suporte gráfico para a criação de modelos orientados a objetos. O uso de formalismos é motivado pela diminuição de ambigüidades, ganhos na consistência dos diagramas e especificações, correção por provas formais, e refinamento de um modelo mais abstrato para um modelo implementacional correto [EVA99].

Os métodos formais podem ser aplicados em graus variados de rigor. Podem ser aplicados em estágios selecionados ou em todos os estágios do ciclo de vida do sistema, e para alguns ou todos os componentes e propriedades do sistema.

Os métodos formais podem ser utilizados em dois níveis no desenvolvimento de sistemas [SOM05]:

- Uma especificação formal do sistema pode ser desenvolvida e matematicamente analisada quanto a inconsistências. Esta técnica é eficaz para descobrir erros e omissões de especificação.
- Pode ser desenvolvida uma verificação formal de que o código de um sistema de software é consistente com a especificação. Isso requer uma especificação formal e é eficaz em

descobrir erros de programação e alguns erros de projeto.

O uso de métodos formais para o desenvolvimento de sistemas conduz a sistemas mais confiáveis e mais seguros [CLA96]. Não há dúvida de que uma especificação formal de sistema apresenta menor probabilidade de conter anomalias que tenham de ser resolvidas pelo projetista do sistema. Contudo, a especificação formal e a prova não garantem que o software seja confiável no uso prático.

Os métodos formais preocupam-se fundamentalmente com os requisitos funcionais do sistema que se deseja especificar. Apesar de ser evidente a necessidade de utilização dos métodos formais, sua ampla utilização entre os desenvolvedores ainda é baixa, devido a fatores tais como [ALE99]:

- Dificuldade de entendimento das notações.
- Dificuldade na formalização de certos aspectos de requisitos.
- Conservadorismo dos desenvolvedores de software.

Os métodos formais permitem que os engenheiros possam capturar e especificar requisitos de forma mais cuidadosa e precisa, reduzindo o custo de desenvolvimento e melhorando a qualidade dos softwares.

Uma especificação nada mais é do que um modelo de um sistema, em um determinado nível de abstração, de forma a ser possível descrever não só o comportamento do sistema, mas também os tipos de dados, caso seja necessário.

Os métodos formais modelam matematicamente o comportamento do sistema, quase sempre o comportamento funcional, procurando reduzir inconsistências e ambigüidades, através das provas de propriedades e animações. Neste caso, métodos formais ajudam a precisar os métodos informais e semi-formais, garantindo especificações mais corretas e seguras. Os métodos formais permitem que o engenheiro de software especifique, desenvolva e verifique o sistema a ser desenvolvido, através da aplicação de uma rigorosa notação matemática, possibilitando assegurar consistência, completude e corretude [ALE99]. Quando os métodos formais são usados durante o desenvolvimento permitem descobrir e corrigir, mais fácil e rapidamente, os problemas existentes.

Uma especificação deve explicitar, de forma exata e não ambígua, todas as condições a que um objeto deve satisfazer. A especificação formal constitui-se numa coleção de diferentes técnicas, mas tendo em comum o uso da matemática para especificar o comportamento dos sistemas. Os custos e benefícios de uma especificação formal variam segundo o modo de utilizá-la, por exemplo, o modo mais simples e uso mais completo é o de utilizá-la como meio para a documentação e comunicação dos requisitos, com o intuito de reduzir ambigüidades [ALE99].

Alguns aspectos básicos comuns que caracterizem uma boa descrição formal são:

- **Fundamentação Matemática:** deve existir um modelo matemático formal que sustente os elementos de definição e de especificação da técnica formal, permitindo que haja possibilidade de análise e verificação de especificações, prototipagem e implementação de testes de conformidade dos sistemas especificados;
- **Descrição Mínima e Precisa:** deve descrever de forma precisa todas as propriedades, de forma a se ter uma descrição com alto grau de abstração;
- **Poder de Expressão:** deve possuir capacidade de expressar de forma concisa, uma grande quantidade de propriedades importantes;
- **Flexibilidade:** fornecer condições para que haja modularidade, de forma a permitir pequenas alterações nas propriedades a serem especificadas;
- **Facilidade de Entendimento:** prover descrições facilmente compreensíveis.

Uma linguagem de especificação precisa e formal é composta por [ALE99]:

- **Sintaxe** que define a notação própria da especificação e que permite que os requisitos sejam interpretados de forma única;
- **Semântica** que ajuda a definir um universo de objetos que serão utilizados na descrição do sistema;
- **Conjunto de restrições** que define as regras que indicam que objetos satisfazem a especificação.

O conceito de software confiável é relativo e está diretamente ligado aos conceitos de corretude e robustez. Corretude pode ser definido como a habilidade de um programa em cumprir o que foi definido em sua especificação. Se não possuímos uma clara noção das atribuições de um programa não temos como julgar se este está correto. Robustez é a capacidade de reagir a situações não incluídas na especificação evitando assim que ocorram estados inconsistentes.

A verificação é uma atividade indispensável em qualquer processo de desenvolvimento de software. É por meio dela que são obtidas evidências da corretude do sistema desenvolvido. As técnicas de verificação de software são usadas para detectar defeitos difíceis de serem identificados pelo desenvolvedor, objetivando a eliminação do maior número de defeitos possíveis [TRA01].

As técnicas de verificação de software podem ser divididas em duas classes: verificação dinâmica e verificação estática. Na **verificação dinâmica**, o código do programa deve ser executado para que se possa verificá-lo. Testes e avaliação de asserções no código são exemplos de verificação dinâmica. No caso das asserções, os resultados da verificação são reportados à medida que o sistema é executado. Assim, é possível que comportamentos indesejados sejam descobertos somente quando o cliente utilizar o sistema. Outra desvantagem desta técnica é que o código referente à funcionalidade verificada deve estar

totalmente implementado.

Na **verificação estática**, a verificação é feita sem que o código seja executado, ou seja, de forma estática. Exemplos de verificação estática são as verificações sintática e semântica e a verificação de modelos, na qual um modelo abstrato do sistema é executado. Nesta classe, o sistema não precisa estar completamente implementado, o que torna os processos de implementação e verificação mais independentes um do outro. Contudo, os modelos abstratos podem não representar corretamente o comportamento dos sistemas correspondentes face à distância sintática e semântica geralmente presente entre a linguagem de modelagem e a de programação.

Bertrand Meyer definiu o conceito de Design by Contracts (programação por contratos) baseado em algumas metodologias utilizadas atualmente: especificação formal, uso intensivo de teste e reutilização de software. As principais idéias deste conceito são [MEY97]:

- **Especificação:** Tentativa de incluir a especificação formal como parte intrínseca da linguagem fazendo assim com que ela não exija esforço adicional.
- **Documentação automática:** A especificação como parte do código podendo ser extraída automaticamente deste.
- **Eliminação de erros:** verificação durante a fase de produção se as circunstâncias especificadas foram de fato cumpridas. Essa verificação pode ser feita através da inclusão de contratos entre os componentes do software.

Design by Contracts sistematiza o uso de diversas técnicas pré-existentes e tenta trazer a especificação e a documentação do código para dentro do mesmo. Um contrato é uma especificação do comportamento de uma classe e seus métodos associados. O contrato esboça os direitos e as responsabilidades de ambos: o método e o cliente do método. Quando quaisquer das responsabilidades não são atendidas, existe uma quebra de contrato, esta indica a existência de um erro em algum lugar na implementação do programa. Contratos contribuem para o reuso e melhoram a robustez do programa. Contratos corretamente implementados reduzem a chance que erros possam passar despercebidos durante o teste de um programa [MEY97].

Os principais elementos para a definição de um contrato são pré-condições, pós-condições e invariantes (o conceito desses elementos será definido na seção 2.3, onde serão explicados os conceitos da linguagem OCL). A forma encontrada para converter as verificações em código foi através do uso intensivo e coordenado de asserções. Um contrato pode ser completamente especificado pelo uso de *pré-condições, pós-condições, e invariantes*. Contratos especificados desta maneira produzem classes que são fácil de usar e entender. O contrato serve como um guia a uma classe como também uma especificação formal do comportamento da classe. Contratos são úteis durante o desenvolvimento e fases de implementação de um programa. A programação por contrato produz classes cujo

comportamento é bem definido e que podem ser reutilizadas [PLO97].

A programação por contratos possibilita em qualquer fase do ciclo de produção verificar se um trecho de software está realizando o que seria esperado. Assim diminui-se a quantidade de erros existentes e a sua correção torna-se muito mais rápida, tornando todo o ciclo de controle de qualidade mais eficaz.

Um fator negativo da programação por contrato é o esforço gasto numa fase inicial do sistema para especificar as regras, utilizações e relacionamentos das classes e também os seus contratos.

2.2 A Linguagem de Modelagem Unificada (UML)

A modelagem é uma tarefa fundamental nas atividades iniciais de desenvolvimento do software, pois é necessário comunicar a estrutura e comportamento desejado para o sistema, visualizar e controlar a arquitetura do software, proporcionar melhor entendimento do sistema, gerenciar os riscos, etc [RAM03]. Para modelar os sistemas é preciso utilizar uma linguagem expressiva, simples, comum e padrão. A UML é uma linguagem para especificação, visualização, construção e documentação de artefatos de sistemas de software. A UML abrange a modelagem estrutural e comportamental dos sistemas.

As propriedades de uma especificação precisa – ausência de ambigüidade, consistência e completude – constituem-se nos objetivos de todos os métodos de especificação. A especificação da UML usa uma combinação de linguagens: um subconjunto de UML, uma linguagem de restrições (OCL) e linguagem natural a fim de descrever a semântica e a sintaxe abstrata da UML [ALE99].

A linguagem UML possui um documento controlado pela OMG (Object Management Group) [UML99], que constitui uma parte importante na definição padrão da linguagem. A abordagem utilizada no documento é a de fornecer uma descrição em meta-modelo da linguagem, sendo apresentada em termos de três visões:

- **Sintaxe abstrata**, expressa usando-se um subconjunto das notações de modelagem estática de UML;
- **Um conjunto de regras de boa formação**, expressas na Linguagem de Restrições de Objetos (OCL);
- **Semântica dos elementos de modelagem**, descrita em linguagem natural.

A UML é atualmente um padrão para especificar, visualizar, documentar e construir artefatos de um sistema. Ela pode ser utilizada em todos os processos, ao longo do ciclo de

desenvolvimento e, por meio de diferentes técnicas de implementação. Não se trata de uma metodologia, mas de uma linguagem de modelagem que possui um meta-modelo bem definido[JAC99].

A Linguagem de Modelagem Unificada inclui [UML99]:

- Elementos do modelo, que capturam os conceitos fundamentais da modelagem e semântica.
- Uma notação para a visualização dos elementos de um modelo.
- Regras que descrevem o uso.

2.2.1 O Meta-modelo da UML

O meta-modelo da UML é um modelo de dados usado para descrever os projetos expressos em UML. Os constituintes deste modelo são os elementos UML que podem ser utilizados para descrever o projeto, como no exemplo da Figura 2 [RAS06].

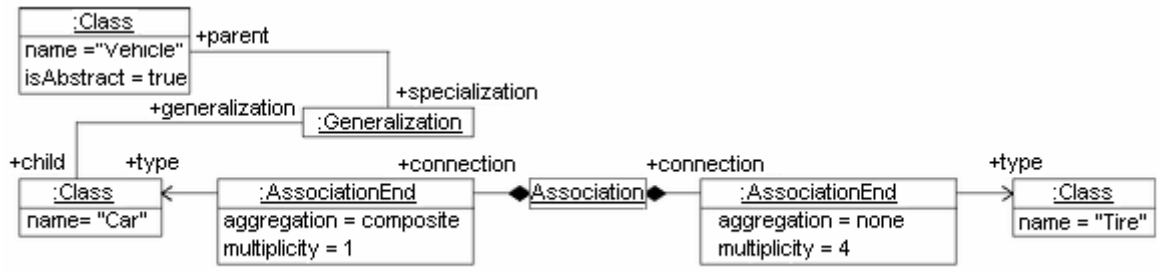

Figura 2 – Projeto baseado no modelo de semântica UML [RAS06].

Existem vários tipos de elementos, um tipo de elemento para cada conceito de modelagem. Os elementos que descrevem o projeto no exemplo acima possuem tipos para representar classes, generalizações, associações e associações finais.

O modelo semântico representa cada elemento UML como um objeto. Ele descreve quais atributos um elemento pode ter e quais relacionamentos que podem existir entre os elementos. Um projeto de software em UML descreve o modelo de um sistema de software. O modelo semântico UML é um modelo usado para descrever modelos. Desta forma, o modelo semântico UML é freqüentemente referenciado como o meta-modelo UML. O meta-modelo é uma linguagem para descrever modelos. Na Figura 3, *Element, ModelElement, Namespace, GeneralizableElement*, etc., são classes no meta-modelo UML. Estas classes são chamadas

meta-classes. Qualquer projeto expresso em UML pode ser representado por instâncias destas classes, suas propriedades e *links* entre elas [RAS06].

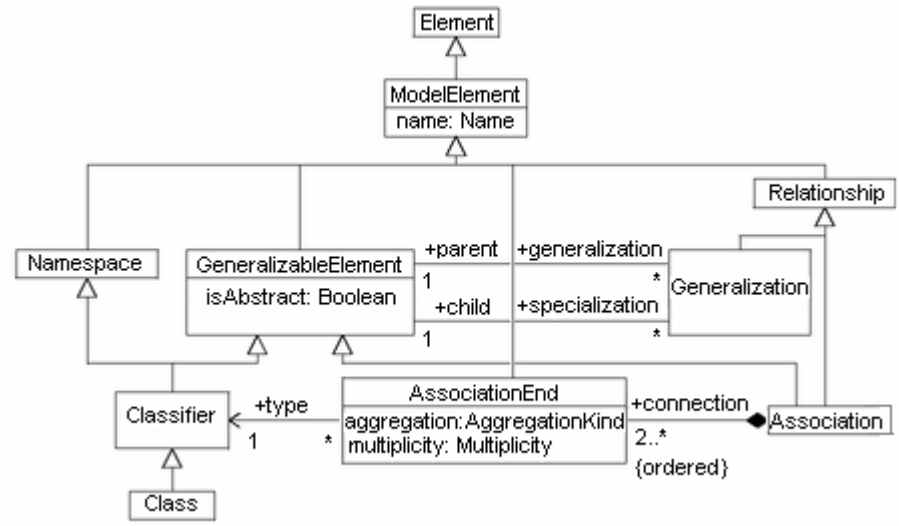

Figura 3 – Modelo Semântico UML [RAS06]

A arquitetura do meta-modelo possui um framework conceitual baseado em quatro camadas. A UML é baseada numa arquitetura de quatro camadas, cada camada sendo uma instância da sua camada superior, sendo a última camada uma instância dela mesma [KLE06]. Estas camadas estão na Tabela 1.

Tabela 1 – Camadas do meta-modelo [OLI04].

Camada	Descrição	Exemplo
Meta meta-modelo	Define a linguagem de especificação de meta-modelos	MetaClass, MetaAttribute, MetaOperation
Meta-modelo	Define a linguagem de especificação de um modelo	Classe, Atributo, Operação, Componente
Modelo	Define a linguagem que descreve a informação de um domínio	Figura, Janela, Mover
Objetos do usuário	Define a informação de um domínio	<Círculo, 10, Azul>

Um modelo é uma representação concreta do projeto de um software. O modelo descreve o domínio da informação dos dados do usuário, os dados do usuário são instâncias do modelo.

Um meta-modelo é um modelo que representa a estrutura e a semântica de um conjunto particular de modelos. O meta-modelo descreve o significado do conjunto de modelos, por exemplo, o meta-modelo da UML é a linguagem utilizada para descrever modelos e um modelo é uma instância do meta-modelo.

Um meta meta-modelo é um modelo que representa como um conjunto particular de meta-modelos é interpretado. O meta meta-modelo é a linguagem utilizada para descrever o meta-modelo da UML, e o meta-modelo da UML é uma instância do meta meta-modelo [KLE06].

Os criadores da UML escolheram uma arquitetura de quatro camadas porque ela fornece a base para alinhar a UML com outros padrões baseados em uma infra-estrutura similar, como a amplamente utilizada Meta-Object Facility (MOF) [KLE06]. Embora não exista um mapeamento direto entre todos os elementos do meta meta-modelo da MOF e os elementos do meta-modelo da UML, existe interoperabilidade entre os dois modelos: o meta-modelo do core package da UML e a MOF são muito similares. Assim, o meta-modelo da UML (um conjunto de diagramas de classe) é um modelo UML.

Os diagramas propostos em UML 1.5 são:

- Diagrama de Caso de Uso
- Diagrama de Classe
- Diagrama de Objeto
- Diagrama de Seqüência
- Diagrama de Colaboração
- Diagrama de Estados
- Diagrama de Atividade
- Diagrama de Componente
- Diagrama de Implantação

A UML fornece fundamentos para a modelagem das partes estruturais e comportamentais de um sistema. A UML cobre as quatro principais dimensões da modelagem de software: funcional (diagramas de casos de uso expressam os requisitos), estática (diagramas de classe), dinâmico (diagramas de estado, diagramas de seqüência para a especificação dos aspectos comportamentais) e físico (diagramas de implementação) [SUN01].

Neste trabalho abordaremos apenas o uso dos diagramas de classe. O Diagrama de

Classes apresenta as classes com os atributos e operações que cada uma de suas instâncias possui, arranjadas em hierarquias que compartilham estrutura e comportamento que estão associados a outras classes. São também levados em consideração os conceitos fundamentais de estruturação que incluem dentre outros a agregação e a generalização [UML99].

Em UML são definidos elementos cujas notações podem ser empregadas nos diversos diagramas, sendo os principais: pacote, estereótipo e anotação. Pacote é um mecanismo para organizar elementos do modelo em grupos, tendo por base as categorias de classe. Estereótipo significa a introdução de novos elementos no meta-modelo, para permitir que os usuários estendam a capacidade de modelagem da linguagem. Anotação é um comentário, colocado em um diagrama, sem qualquer conteúdo semântico [ALE99].

A UML foi definida através de uma abordagem de meta-modelo para tornar sua sintaxe mais precisa. Um meta-modelo para uma linguagem é uma definição da linguagem escrita em termos da mesma. Um modelo UML é considerado sintaticamente correto se ele é uma instância consistente do meta-modelo da UML. A consistência desta instância é garantida através da estrutura do meta-modelo, por exemplo, multiplicidades nas associações, e um conjunto de regras de boa formação expressadas em OCL, que são restrições lógicas nos elementos dos modelos. Por exemplo: para um modelo UML estar correto não deve existir nenhuma herança cíclica e um estado final não deve ter nenhuma transição de saída.

2.3 Object Constraint Language - OCL

A linguagem OCL (Object Constraints Language) começou a ser desenvolvida por volta de 1995 por Jos Warmer e Steve Cook, na IBM, como uma linguagem de modelagem de negócio, porém só foi formalmente definida como parte do padrão UML em 1997, pelo OMG (Object Management Group) [WAR99]. Os usuários de UML e outras linguagens de modelagem podem utilizar OCL para especificar restrições e outras expressões vinculadas a cada modelo.

OCL é uma linguagem de expressões textuais precisas, utilizada na descrição de restrições em modelos orientados a objeto, como forma de complementar a parte gráfica dos modelos, para descrever restrições, que não conseguem ser diagramaticamente representadas. OCL é uma linguagem formal baseada em modelos, é uma linguagem que adota uma sintaxe simples, não simbólica, e que utiliza símbolos matemáticos mais simples da teoria de conjuntos e lógica, numa proposta de ser precisa, porém de fácil compreensão (escrita-leitura) por quem não possui conhecimento matemático aprofundado.

Um dos mais importantes aspectos de OCL é que se tornou parte do padrão UML

para especificação [JAC99, BOO99, RUM99]. O seu objetivo foi o de acrescentar à definição de UML a possibilidade de especificar restrições aos modelos UML. Restrições são limitações em um ou mais valores de parte ou do todo de um objeto, em um modelo orientado a objeto ou de um sistema. Na prática são detalhes e informações extras, que não conseguem ser expressos pelos modelos gráficos. Antes de OCL estas restrições eram descritas em linguagem natural, resultando em ambigüidades e imprecisões.

A UML não dispõe de recursos expressivos para especificação de restrições sobre um modelo orientado a objetos, nesse contexto surgiu OCL [MEN02]. Uma expressão é uma indicação ou especificação de um valor. Uma restrição pode aparecer em um ou mais valores de um modelo ou sistema orientado a objetos. As expressões definem tipicamente as condições que se deve assegurar para o bom funcionamento do sistema que está sendo modelado, ou ainda, a modelagem de informações contidas nos objetos descritos num determinado modelo. Quando uma expressão OCL é avaliada, não ocorre qualquer efeito colateral, isto é, o estado de execução do respectivo sistema não é alterado. As expressões podem ser utilizadas para especificar as restrições de operações ou ações de um determinado objeto, que quando executadas alteram o estado do sistema [0CL06]. Não se trata de uma linguagem de execução nem de verificação de restrições. Contudo, suas expressões podem ser verificadas antes da execução do modelo, ainda durante a modelagem [MEN02].

A utilização da idéia de restrições acrescenta algumas vantagens, a partir das quais se pode destacar [0CL06]:

- **Melhor Documentação:** por acrescentar aos modelos, informações sobre os elementos e seus relacionamentos. Um modelo gráfico pode conter algumas restrições, como é o caso da multiplicidade nos relacionamentos; no entanto, alguns detalhes não são possíveis de serem representados.
- **Precisão Aumentada:** dado que as restrições não podem ser interpretadas de forma diferente por várias pessoas, são não ambíguas, tornando o modelo ou sistema, sobre o qual se aplicam, mais preciso.
- **Comunicação sem Enganos:** por causa da precisão na descrição das informações, os desenvolvedores são capazes de comunicar as suas intenções de forma não ambígua, descobrindo antecipadamente possíveis defeitos.

Segundo essa classificação, OCL é uma linguagem declarativa, dado que as restrições não conduzem a efeitos colaterais, ou seja, o estado do sistema não muda em função da avaliação de uma expressão em OCL, embora essa possa ser utilizada para especificar uma pós-condição. Com isso, podem-se destacar as seguintes vantagens:

- O modelador não precisar decidir como manipular a quebra de uma restrição.
- As restrições não mudariam ao longo do tempo.

- A checagem de ocorrência de quebra de restrição é feita atomicamente.

Desta forma, a especificação de restrições é feita a nível conceitual, onde os aspectos de implementação são considerados como irrelevantes.

As expressões OCL definem as condições que devem assegurar o funcionamento de um sistema que será modelado, ou a modelagem de informações contidas nos objetos descritos num determinado modelo [OLI01]. As expressões podem ser utilizadas para especificar as restrições de operações ou ações de um determinado objeto, que quando executadas alteram o estado do sistema.

2.3.1 Descrição da Linguagem

OCL baseia-se em modelos, é uma Linguagem de Restrições a Objetos utilizada para complementar os modelos orientados a objeto [WAR99]. Neste trabalho foi escolhida esta linguagem pelo seguinte motivo: OCL faz parte do padrão de modelagem orientado a objeto UML, em uso por toda a comunidade de orientação a objetos [JAC99].

2.3.2 Tipos de Restrições

OCL permite especificar quais estados são válidos e quais mudanças de estado irão ocorrer como resultado de uma operação específica através, por exemplo, de uma pós-condição.

Todas as expressões em OCL têm um tipo e avaliam um valor quando aplicadas a um contexto específico. Os usos mais comuns de OCL são em [OCL06]:

- **Invariantes para classes:** um invariante é uma condição que deve ser válida durante todo o tempo pelos objetos daquela classe.
- **Invariantes para esterótipos:** ele é usado quando da definição de um estereótipo para especificar uma condição que deve ser aplicável a todas as classes daquele estereótipo.
- **Pré e pós-condições para operações:** uma pré-condição é uma restrição que especifica o que deve ser verdadeiro antes de uma operação de uma classe ser executada. Uma pós-condição é uma restrição que especifica o que deve ser verdadeiro depois que uma operação é executada.
- **Regras navegacionais num modelo:** OCL pode ser usada para especificar uma navegação. Começando de um objeto específico, pode-se navegar através de uma

associação sobre o diagrama de classe para se referir a outros objetos e suas propriedades.

- **Regras de derivação:** uma regra de derivação especifica como um valor específico é calculado a partir de outros valores. Por exemplo, como um atributo é derivado de outros valores de outros atributos (ex.: a idade pode ser derivada da data de nascimento e da data atual).
- **Guardas:** uma guarda é uma condição que especifica quando se deve executar uma atividade específica (ou processo) ou, quando várias alternativas existem, qual das alternativas deve ser executada. Também podem ser usadas guardas em diagramas de estado para mostrar sobre quais condições uma transição de estado acontece, ou num diagrama de seqüência ou colaboração para especificar quando uma mensagem específica deve ser enviada. Uma guarda é uma expressão OCL que, quando avaliada verdadeira, indica que a mensagem, a transição de estado, ou a atividade daquela guarda deve ser executada ou enviada.

2.3.3 Contexto de uma Expressão OCL

A definição de contexto de uma expressão OCL especifica o modelo entidade para o qual a expressão OCL é definida. Geralmente é uma classe, interface, tipo de dados, ou componente. Em termos dos padrões UML, isto é chamado de um *Classificador* [OCL06].

Às vezes o modelo entidade é uma operação ou atributo, e raramente é uma instância. É sempre um elemento específico do modelo, geralmente definido em um diagrama UML. Este elemento chama-se o *contexto* da expressão. O tipo contextual é o tipo do contexto, isto é importante porque as expressões OCL são avaliadas para um objeto simples, que é sempre uma instância do tipo contextual. A instância contextual serve para distinguir entre o contexto e a instância para a qual a expressão é avaliada. Às vezes é necessário para se referir explicitamente à instância contextual, para isto utiliza-se a palavra *self*.

2.4 Frameworks

Frameworks orientados a objetos são coleções de classes organizadas em uma arquitetura abstrata com o objetivo de implementar uma família de problemas relacionados [JOH97] [FAY99] [JOH98]. É extensa a literatura sobre o uso de frameworks, o que mostra sua importância como uma ferramenta de reuso. Eles podem ser vistos como aplicações

incompletas que devem ser especializadas para o desenvolvimento de aplicações concretas. Uma outra característica importante desta tecnologia é que o fluxo de controle da aplicação é invertido, isto é, não é responsabilidade do engenheiro de aplicações determinar esse fluxo, mas do próprio framework [PRE99]. A grande vantagem desta abordagem é a promoção de reuso de código e projeto, que pode diminuir o tempo e o esforço exigidos na produção de software.

Os conceitos de framework têm sido empregados com sucesso como ferramentas para obtenção de reuso de software. Eles reduzem o esforço de desenvolvimento e aumentam a qualidade dos sistemas de software produzidos. Conforme os frameworks foram se tornando populares, suas fraquezas foram aparecendo. Em particular, o processo de instanciação de frameworks é uma tarefa difícil porque os desenvolvedores da aplicação precisam entender os detalhes do projeto do framework, tornando este processo lento e custoso [OLI05].

A reutilização em um framework orientado a objetos é a combinação de uma arquitetura orientada a objetos semi-completa com incrementos específicos da aplicação para satisfazer as necessidades da aplicação[OLI01].

Frameworks orientados a objetos permitem a construção rápida e fácil de um software [FAY99]. A demanda por este tipo de tecnologia é cada vez mais constante. A tecnologia dos frameworks orientados a objetos permite mudanças rápidas de desenvolvimento para que o software possa se adequar às alterações necessárias. Eles geram aplicações instanciadas com muita rapidez, acompanhando assim a constante necessidade de gerar serviços e produtos inovadores. As suas principais características são:

- **Reutilização:** a possibilidade de reutilizar código nas várias aplicações instanciadas pelos frameworks orientados a objetos é que permite diminuir o tempo de desenvolvimento do software. Utilizando uma modelagem orientada a objetos reutilizável, frameworks permitem o reuso das classes, subsistemas ou até o sistema para a geração de novas aplicações instanciadas [FAY99].
- **Flexibilidade:** frameworks orientados a objetos são flexíveis o suficiente para construir variações e extensões no software. Isso permite desenvolver uma gama de aplicações que cobrem um domínio bem amplo. Frameworks orientados a objetos possuem um núcleo de software, com trechos de código já escritos, e vários pontos de flexibilização, que necessitam de desenvolvimentos futuros. Nesses pontos de flexibilização são implementadas as variações e extensões necessárias para a criação da aplicação final [FAY99].

Para o desenvolvimento do núcleo é preciso que se faça, antes de qualquer coisa, uma análise de requisitos das várias aplicações pertencentes ao domínio. Nesta análise de requisitos são identificados os pontos em comum das aplicações e as variações. No desenvolvimento dos frameworks orientados a objetos são implementados os vários pontos

em comum, e indicados os locais onde implementar os vários pontos de flexibilização.

2.4.1 Instanciação de Frameworks

O processo de reutilização de frameworks é chamado de processo de instanciação. É durante este processo que os pontos de extensão existentes no framework são preenchidos para se obter a aplicação final [OLI04]. Os pontos de extensão permitem a uma aplicação estender as interfaces estáveis dos frameworks.

Foi definida uma linguagem de domínio (Domain Specific Language - DSL) que representa as atividades relacionadas ao domínio de instanciação de frameworks de forma textual [OLI04]. A esta DSL foi dado o nome RDL (Reuse Description Language). RDL é uma linguagem que foi criada para representar explicitamente as atividades de instanciação de frameworks. A RDL trabalha com modelos UML, através da transformação de modelos para produzir instâncias de aplicações válidas.

Na instanciação o processo de desenvolvimento de uma aplicação deve ser modificado para incorporar nesta aplicação as características do framework. Este processo inicia com a fase de requisitos, onde são identificados e expressos em uma notação específica os requisitos funcionais e não funcionais. Em seguida o domínio da aplicação é analisado para se produzir um modelo conceitual do problema. Após a escolha do framework, o modelo conceitual da aplicação é integrado/adaptado com o modelo de classes presente no framework. Na seqüência ocorre a codificação e teste.

Para se obter a sistematização do processo de reutilização, é necessário representar as atividades envolvidas na abordagem. Uma adaptação típica de framework tem duas fases [OLI05]:

- Entendimento do projeto do framework.
- Entendimento dos pontos de extensão do framework de acordo com requisitos específicos para produzir os incrementos da aplicação.

Inicialmente identifica-se a presença de dois atores principais: o reutilizador e o projetista da reutilização [OLI04]. O primeiro desempenha o papel do desenvolvimento baseado em reutilização e tem como finalidade aprimorar a construção do software através da reutilização. O segundo desempenha o papel do desenvolvimento para reutilização e tem como finalidade desenvolver o artefato reutilizável e toda a documentação de reutilização associada.

Na Figura 4 está representado o processo de reutilização, ele é caracterizado pelas

atividades de busca, integração e adaptação. Estas atividades se integram com as atividades do processo tradicional de análise de domínio, design e implementação e introduzem o conceito de reutilização.

Como primeiro passo, o reutilizador deve fazer uma especificação inicial da aplicação a ser desenvolvida. Esta especificação deve identificar as possíveis soluções para o problema.

Depois da especificação inicial, o reutilizador busca em um repositório artefatos reutilizáveis para verificar se existe algum que se adapte à aplicação em desenvolvimento. Após esta busca, o reutilizador inicia a integração e adaptação.

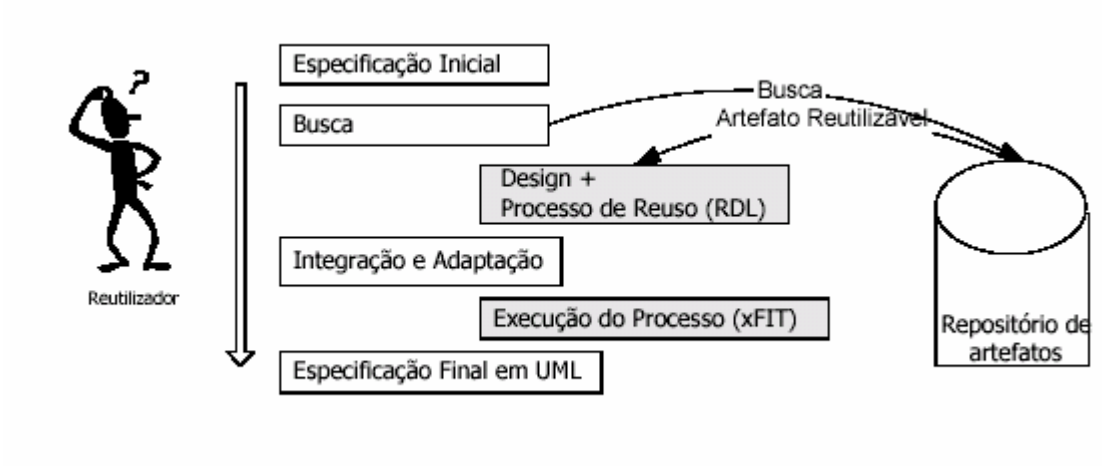

Figura 4 – Processo de Reutilização [OLI04].

A estrutura do artefato reutilizável é composta por dois tipos de documentos: uma especificação em UML e um programa em RDL (*CookBook*). A especificação em UML representa o design do artefato reutilizável através de diagramas de classes com a extensão de UML-FI (Framework de Instanciação UML), que tem como objetivo especificar de forma mais precisa os elementos relevantes para o processo de instanciação. O programa em RDL especifica o processo de reutilização.

A abordagem para o processo de reutilização é um ambiente que tem como entrada o artefato reutilizável. Este ambiente executa o C*ookbook* que comanda a manipulação da representação em XMI, para introduzir a parametrização necessária [OLI04].

Uma vez terminado o processo de reutilização, o ambiente gera um documento em XMI que pode ser re-introduzido nas ferramentas CASE para uma eventual geração de código ou visualização do design final.

No desenvolvimento de uma aplicação baseada em frameworks o reutilizador deve preencher os pontos de extensão / flexibilização existentes com as características específicas da aplicação.

2.5 Conclusões

Este capítulo apresentou o conceito de métodos formais, que são um conjunto de ferramentas e notações utilizadas para especificar de forma não ambígua os requisitos de um sistema e fornecem provas da corretude da implementação. Também foi apresentado o meta-modelo da UML que é um modelo de dados que representa a estrutura e a semântica de um conjunto particular de modelos e é usado para descrever os projetos expressos em UML. A especificação da UML usa uma combinação de linguagens para descrever a semântica e a sintaxe abstrata da UML:

- subconjunto de UML – subconjunto das notações de modelagem estática de UML.
- linguagem de restrições – Linguagem de Restrições de Objetos (OCL).
- linguagem natural – para descrever a semântica dos elementos de modelagem.

Assim como a UML, que possui um meta-modelo bem definido e um conjunto de restrições que melhoram a qualidade dos modelos, este trabalho tem o intuito de definir o meta-modelo da linguagem RDL. Para isto, será utilizado um diagrama de classes UML, compatível com o meta-modelo da UML e será definido um conjunto de restrições em linguagem natural e em seguida mapeadas para a linguagem de restrição de objetos para garantir a consistência dos modelos criados a partir de RDL.

Também foi introduzida a linguagem OCL, que será utilizada na definição das restrições do meta-modelo da linguagem RDL; foi abordado o conceito de frameworks orientados a objetos e a Linguagem de Descrição de Reuso, que será apresentada com mais detalhes no Capítulo 3.

3 LINGUAGEM DE DESCRIÇÃO DE REUSO - RDL

RDL é uma linguagem que permite a especificação da ordem e as dependências de estado nas atividades de programação orientada a objetos que são freqüentemente utilizadas na instanciação de um projeto incompleto.

RDL é uma linguagem que tem o objetivo de fornecer mecanismos para os desenvolvedores de framework para representar explicitamente as tarefas de instanciação. RDL é uma linguagem de programação e independente de domínio de framework e manipula elementos do projeto expressos em UML. As abstrações da RDL foram propostas baseadas na abordagem de cookbook e exploram o uso de design patterns [OLI05]. É importante ressaltar que neste trabalho a linguagem RDL não será detalhada, particularidades da linguagem podem ser encontradas em [OLI07].

A abordagem de instanciação de frameworks consiste em uma linguagem de processos, RDL, que permite que os desenvolvedores de framework representem os passos de adaptação, e xFIT, uma ferramenta de suporte que trabalha com modelos UML transformando um diagrama de classes do framework em diagramas de classe da aplicação baseando-se nas entradas do desenvolvedor da aplicação. A Figura 5 fornece uma visão geral da abordagem [OLI05].

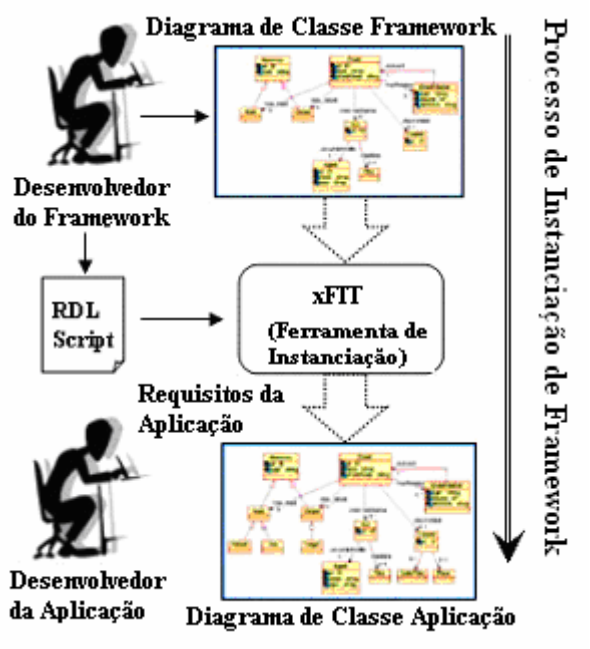

Figura 5 – Processo de Instanciação de Framework [OLI05].

Para instanciar um framework utilizando esta abordagem é necessário seguir os passos:

- O desenvolvedor do framework fornece um diagrama de classes do framework de acordo com o formato XMI (um arquivo representando o modelo).
- O desenvolvedor do framework fornece um script RDL contendo os passos de instanciação do framework.
- O desenvolvedor da aplicação executa xFIT com o script RDL e o diagrama de classes do framework. O desenvolvedor da aplicação fornece feedback de acordo com os requisitos específicos da aplicação.
- No final do processo de geração xFIT irá rodar as tarefas de validação e relatar os erros encontrados. Então é produzido um diagrama de classes incluindo o framework e as classes específicas da aplicação.
- O desenvolvedor da aplicação pode então usar uma ferramenta Case para abrir o modelo da aplicação e testar o resultado, então o processo é encerrado.

RDL representa o reuso de frameworks orientados a objetos, melhorando a maneira que os desenvolvedores de framework representam o processo de instanciação por meio de seqüências de ações de reuso e suas restrições de ordem e estado. RDL permite a especificação de padrões de instanciação, muito utilizado como forma de facilitar o reuso. RDL trabalha com especificações UML. RDL representa as atividades típicas de programação orientada a objetos que atuam nos elementos do projeto, facilitando assim o entendimento [OLI01].

Um Artefato Reutilizável é um conjunto de documentos estruturados desenvolvidos pelo desenvolvedor do framework e requeridos para a instanciação do framework [OLI05]. Estes documentos descrevem uma Representação do Projeto, uma Especificação da Instanciação e um Conjunto de Restrições, como está representado na Figura 6.

A Representação do Projeto é caracterizada por um diagrama de classes, ela contém *placeholders* para extensão do framework em nível de projeto. *Placeholders* são representados usando UML-FI.

A Especificação da Instanciação é expressa em RDL, onde um script permite a especificação de "guias" para o processo de instanciação usando atividades de programação orientada a objetos, como extensão de classes, redefinição de métodos e aplicação de padrões. Além disso, RDL permite a especificação de elementos requeridos para completar uma instanciação específica e a ordem requerida das tarefas de instanciação.

O Conjunto de Restrições é utilizado para descrever um conjunto de restrições estruturais e comportamentais que podem ser verificadas por uma ferramenta para avaliar a integridade do sistema.

É importante mencionar que RDL tem cerca de 20 comandos desenvolvidos para facilitar o reuso. Os comandos RDL são executados pela ferramenta xFIT, uma ferramenta de instanciação de frameworks. xFIT tem como entrada um script RDL e um arquivo XMI. O script RDL representa o processo de reuso e é desenvolvido manualmente pelo desenvolvedor do recurso reutilizável (o desenvolvedor do framework). O arquivo XMI representa o projeto do recurso reutilizável e o código que será customizado pelo reutilizador durante o processo de reuso [OLI01].

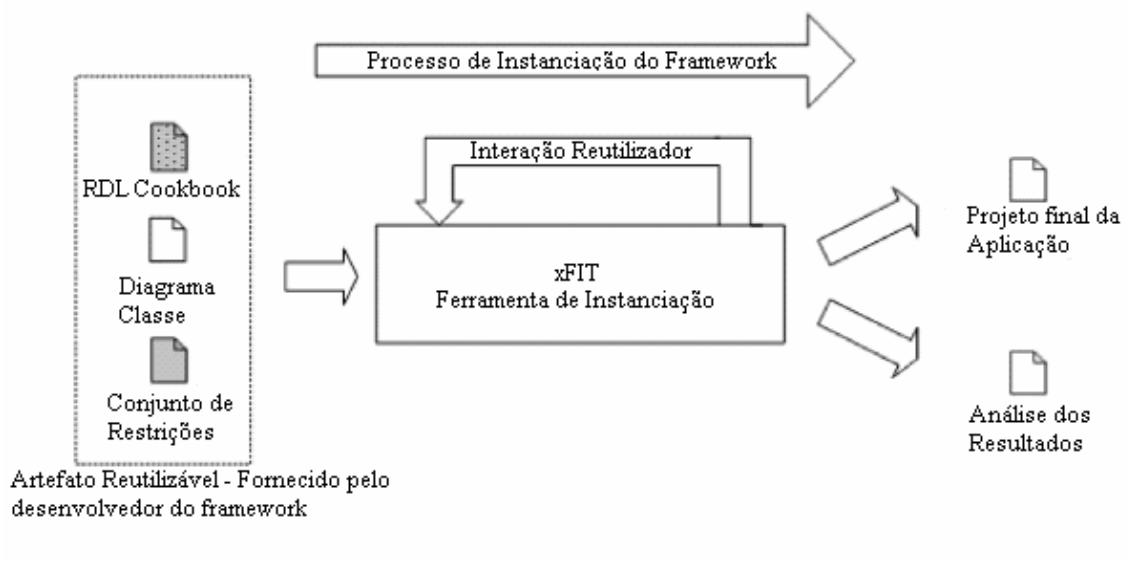

Figura 6 – Visão geral da abordagem [OLI01].

A customização ocorre quando os desenvolvedores da aplicação customizam uma implementação reutilizável em locais especiais, chamados Pontos de Variação, para acomodar as necessidades específicas da aplicação. Um processo de reuso pode ser visto como um conjunto de atividades executadas pelo desenvolvedor da aplicação, o reutilizador. As atividades de reuso são relacionadas e restritas a uma ordem de execução. As atividades de reuso mostram como os pontos de variabilidade são expressos [OLI01]. A natureza das atividades executadas neste processo podem ser descritas como:

- Relacionadas às tarefas de programação orientada a objetos: estendendo classes.
- Seqüenciais: para redefinir um método é necessário estender sua classe.
- Dependentes de estados: a redefinição de um método pode ser dependente da redefinição de outros.
- Baseadas em entradas externas: o nome de uma nova classe deve ser descoberto em tempo de instanciação e não durante a construção do framework.

- Assíncronas: dependendo do tamanho do framework algumas atividades podem rodar em paralelo.
- Baseadas em Padrões: um padrão resolve um problema de instanciação que ocorre recorrentemente.

Um conceito importante nas atividades de reuso são: Atribuição, Padrões e Intervenção do Usuário. Atribuição é uma técnica típica de linguagem de programação utilizada para armazenar dados temporários. Isto é importante porque as atividades de reuso podem criar novos ou modificar elementos existentes e estas modificações podem ser atribuídas a variáveis para utilizar durante o processo de reuso, um exemplo é armazenar o nome de uma nova classe criada. Padrões são muito importantes no processo de reuso. Um padrão agrupa as atividades de reuso como um conjunto de tarefas que devem ser executadas para incluir uma nova característica, ou funcionalidade na aplicação final. Um recurso reutilizável típico é desenvolvido para ser configurado durante o reuso. Durante o desenvolvimento é difícil antecipar os tipos de dados que serão utilizados para customizar o recurso. Por isso é importante definir quais atividades de reuso necessitam da intervenção do usuário e quais dados devem ser fornecidos durante o reuso [OLI01].

RDL é uma linguagem de programação e um framework independente de domínio e manipula elementos do projeto expressos em UML. As abstrações de RDL têm sido propostas baseadas na abordagem de cookbook e explorando o uso de design patterns [OLI05].

As construções em RDL são representadas por *cookbooks*, *recipes* e *patterns*. Um *cookbook* RDL contém um conjunto de *recipes* RDL. *Recipes* possuem tarefas de instanciação relacionadas a aspectos particulares da arquitetura de um framework. Patterns descrevem passos recorrentes de instanciação encontrados durante uma adaptação de framework, por exemplo, design patterns.

A estrutura de um script RDL é a mesma encontrada na maioria das linguagens de programação imperativas, construtores e procedimentos. Entretanto, para seguir a terminologia de instanciação do framework, é utilizado o termo *Cookbook* para representar um programa e *Recipe* para representar um procedimento. Cada *Cookbook* deve possuir uma *Recipe* chamada *main* que identifica o ponto de entrada na execução do processo de instanciação. Em um script RDL, deve haver pelo menos uma recipe chamada *main* representando o ponto inicial do cookbook. Recipes podem chamar umas às outras, recebendo parâmetros e retornando valores de uma maneira similar às funções em linguagens procedurais. A estrutura geral de um script está exemplificada no trecho de código abaixo [OLI07]:

```
COOKBOOK Example
                    RECIPE main
                // fazer alguma coisa
          END_RECIPE;
END_COOKBOOK
```

3.1 Conclusões

Neste capítulo foi apresentada a RDL – Linguagem de Descrição de Reuso – que representa o reuso de frameworks orientados a objetos, melhorando a maneira que os desenvolvedores de framework representam o processo de instanciação por meio de seqüências de ações de reuso e suas restrições de ordem e estado. E foram descritos os próximos passos a serem seguidos no sentido de melhorar a formalização da linguagem.

No Capítulo 4 serão descritos alguns trabalhos relacionados ao tema central desta dissertação: Formalização de Linguagens de Reuso. Serão apresentadas duas abordagens: Actions Semantics e BasicMTL.

4 TRABALHOS RELACIONADOS

Neste capítulo descrevem-se dois trabalhos relacionados a esta pesquisa: Action Semantics e BasicMTL.

4.1 Action Semantics

Action Semantics são utilizadas para manipular elementos UML, como por exemplo, transformação de modelos. A proposta da Action Semantics é fornecer um meta-modelo integrado ao meta-modelo da UML e um modelo de execução. Como um futuro padrão da OMG, a integração da Action Semantics na UML deve facilitar a interoperabilidade de ferramentas e permitir a modelagem e simulação de executáveis, bem como geração de código e casos de teste. Esta abordagem cobre as capacidades de metaprogramação como, por exemplo, refactoring e aplicação de design patterns [SUN01]. A integração com a UML permite que as restrições (pré ou pós condições e invariantes) escritas em OCL sejam aplicadas com Action Semantics. Action Semantics propõe o uso de OCL para verificar se uma dada implementação respeita um conjunto de restrições em OCL; esta proposta compartilha do mesmo objetivo da definição do meta-modelo RDL e do uso de restrições em OCL propostos nesta dissertação.

Action Semantics podem ser usadas no nível de meta-modelo para auxiliar o projetista em atividades como transformações preservando o comportamento, aplicação de design patterns.

A intenção não é propor outra abordagem para transformação de modelos, aplicação de padrão ou refactoring. O objetivo é utilizar Action Semantics como uma linguagem de meta-programação para auxiliar a implementação de abordagens existentes. O principal interesse de utilizar UML/Action Semantics em nível de meta-modelo para expressar estas transformações é que pode-se utilizar os princípios Orientados a Objetos para estruturá-las em Componentes de Transformação Reutilizáveis. Além disso, Action Semantics podem ser combinadas com regras escritas em OCL para verificar se uma transformação (ou conjunto de transformações) pode ser aplicada a um dado contexto.

O objetivo é fazer com que a UML seja uma linguagem de modelagem de modelos executáveis para permitir aos projetistas fazer testes e verificações e se desejável, gerar 100% do código.

A intenção é que Action Semantics se torne um padrão OMG e uma base comum

para todas as linguagens de ação existentes. A Action Semantics proposta é baseada em três abstrações:

- **Meta-modelo:** ele estende o meta-modelo atual. A Action Semantics é integrada no meta-modelo atual e permite uma sintaxe precisa de ações substituindo todos os itens não interpretados.
- **Modelo de execução:** é um modelo UML. Ele permite que as mudanças em uma entidade sejam modeladas.
- **Semântica**: a execução de uma Action é definida precisamente com um ciclo de vida que mostra sem ambigüidade o efeito de executar a ação na instância atual do modelo de execução.

A Figura 7 mostra a inclusão de Action Semantics em UML.

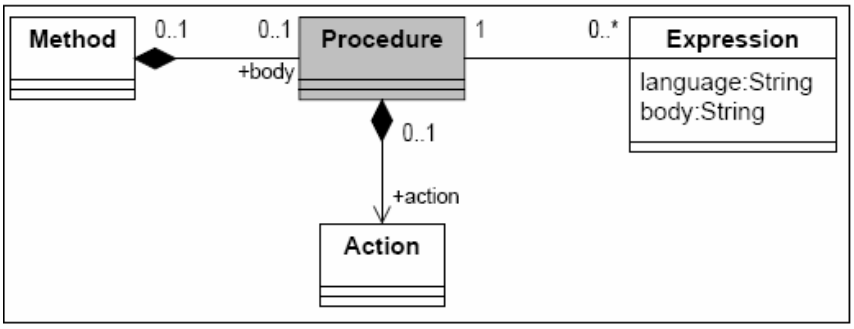

Figura 7 – Action Semantics em UML [SUN01].

A Action Semantics traz algumas possíveis mudanças ao processo tradicional de desenvolvimento de software, sendo um passo importante em direção ao uso da UML em ambiente de desenvolvimento efetivo, porque ela oferece a possibilidade de animar modelos de projeto e refiná-los até a sua implementação.

A abordagem de desenvolvimento inicia com um modelo inicial do projeto, criado pelos projetistas a partir de um modelo de análise. Este modelo é completamente independente do ambiente de implementação. Desde que este modelo contenha Action Semantics, ele pode ser animado e validado. Quando a validação está concluída, os projetistas podem adicionar alguns aspectos específicos do ambiente para o modelo do projeto (acesso à base de dados, distribuição), aplicar design patterns e reestruturar o modelo utilizando refactorings.

A seguir, um exemplo da remoção de uma classe de um pacote. Este código só pode ser aplicado se algumas condições forem verificadas: a classe não está sendo referenciada por qualquer outra classe e não possui sub-classes. Estas condições e as mudanças são definidas

nas Actions Semantics a seguir:

```
Package::removeClass(class: Class)
pre:
    self . allClasses!includes(class) and
    class . classReferences!isEmpty and
    ( class . subclasses!isEmpty or class.features!isEmpty)
actions:
    let aCollection := class .allSuperTypes
    class .allSubTypes!forAll(sub : Class j
    aCollection!forAll(sup : Class j sub.addSuperClass(sup)))
    class . delete
post:
    self . allClasses!excludes(class) and
    class@pre.allSubTypes!forAll(each : Class j
    each.allSuperTypes.includesAll(class@pre.allSuperTypes)) and
    class@pre.features!forAll(feat : Feature j feat .owner = nil)
```

Neste trecho são chamadas duas funções que não estão presentes na Action Semantics, *delete* e *addSuperClass()*, e devem ser definidas em outro lugar: a deleção deve ser efetiva, deletando as características, as associações e generalizações relacionadas à classe. A última característica está descrita abaixo: O objetivo é criar uma generalização entre duas classes:

```
Class :: addSuperClass(class: Class)
pre:
    self .allSuperTypes!excludes(class)
actions:
    let gen := Generalization.new
    gen.addLinkTo(parent, class)
    gen.addLinkTo(child, self)
post:
    self .allSuperTypes()!includes(class)
```

A análise da Action Semantics irá ajudar a verificar se um padrão foi implementado corretamente, mais precisamente será possível verificar se uma dada implementação de um padrão respeita um conjunto de restrições pré-definidas em OCL.

4.2 BasicMTL

A BasicMTL é uma linguagem para transformação de modelos, ela depende de meta-modelos. A linguagem manipula modelos oriundos de qualquer tipo de meta-modelo e em

qualquer tipo de repositório. É possível representar tarefas genéricas independente do nível de abstração, pois a linguagem possui múltipla herança para classes e bibliotecas. Existe o conceito de manipulação de visão, onde visões são modelos virtuais cujos meta-modelos são descritos por uma biblioteca BasicMTL [MTL05].

BasicMTL é uma linguagem tipada que possui tipos estáticos para BasicMTL e tipos implícitos para elementos de modelos. É uma linguagem orientada a objetos baseada nos diagramas de classe UML. A linguagem utiliza OCL em sua sintaxe abstrata. Uma especificidade da BasicMTL é que os modelos a serem manipulados são acessados em tempo de execução. O uso da abordagem de meta-modelos e OCL na sintaxe abstrata da linguagem BasicMTL faz com que os modelos criados com a linguagem sejam mais corretos e confiáveis, da mesma forma como está sendo proposto na linguagem RDL, a definição do meta-modelo RDL e restrições escritas em OCL.

O objetivo da linguagem é melhorar a reutilização de engenharia de software de transformações de larga escala. A transformação de modelos requer longo tempo, o primeiro requisito do usuário é ter um modelo durável e portável. A BasicMTL permite a construção de várias abordagens de framework. Cada usuário tem o seu próprio interesse e a BasicMTL deve estar disponível para organizar as transformações que não foram construídas para o mesmo propósito, mas que possuem algumas similaridades. A idéia é criar um framework de grande escala que organiza isto de alguma maneira [MTL05].

A linguagem possui um conceito principal: existe uma analogia entre as classes definidas pelo programador da transformação e as meta-classes do modelo manipulado. A Figura 8 mostra um objeto BasicMTL e a analogia entre o elemento do modelo.

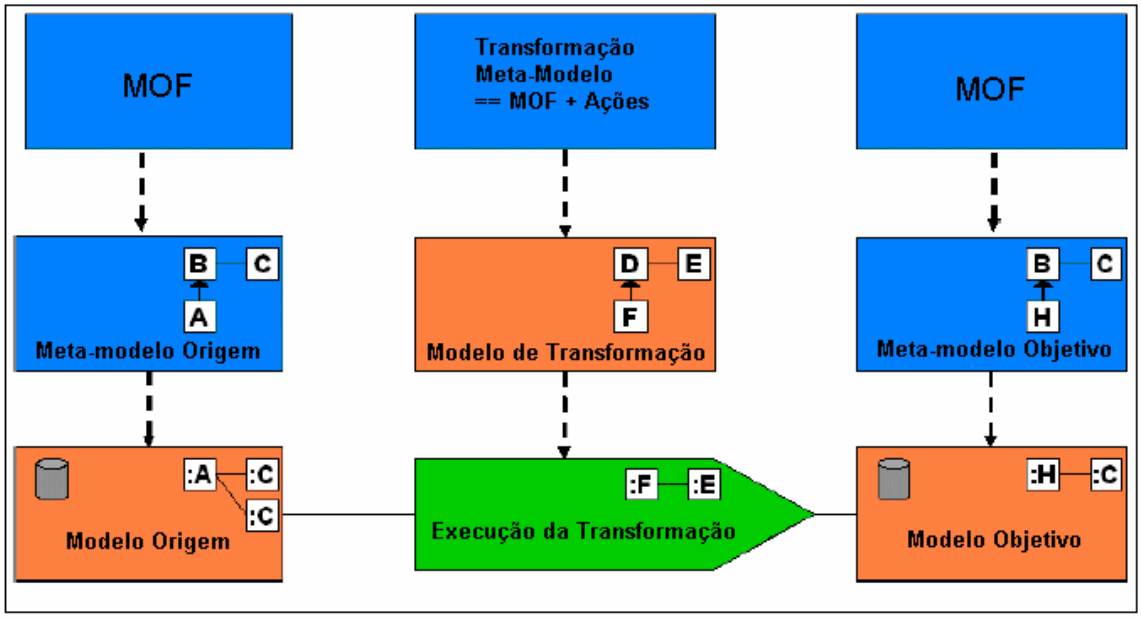

Figura 8 – Objeto BasicMTL – Analogia Elemento Modelo [MTL05].

BasicMTL possui uma sintaxe abstrata que define os conceitos que fazem parte da linguagem e possui um conjunto de restrições definidas utilizando OCL. A sintaxe abstrata é dividida em vários blocos:

- O bloco TipoDeDados descreve os conceitos que definem o tipo do sistema, ele mostra quais são os tipos pré-definidos na linguagem.
- O bloco Propriedade descreve o mecanismo que é usado em várias partes no modelo.
- O bloco Biblioteca descreve o conceito de biblioteca na linguagem.
- O bloco de Classes e Operações descreve o conceito de classes e operações na linguagem.
- O bloco Expressão descreve a estrutura das expressões na linguagem.

A Figura 9 representa o meta-modelo da BasicMTL com relação ao uso de Classes de Usuário (*UserClass*) e Operações (*Operations*).

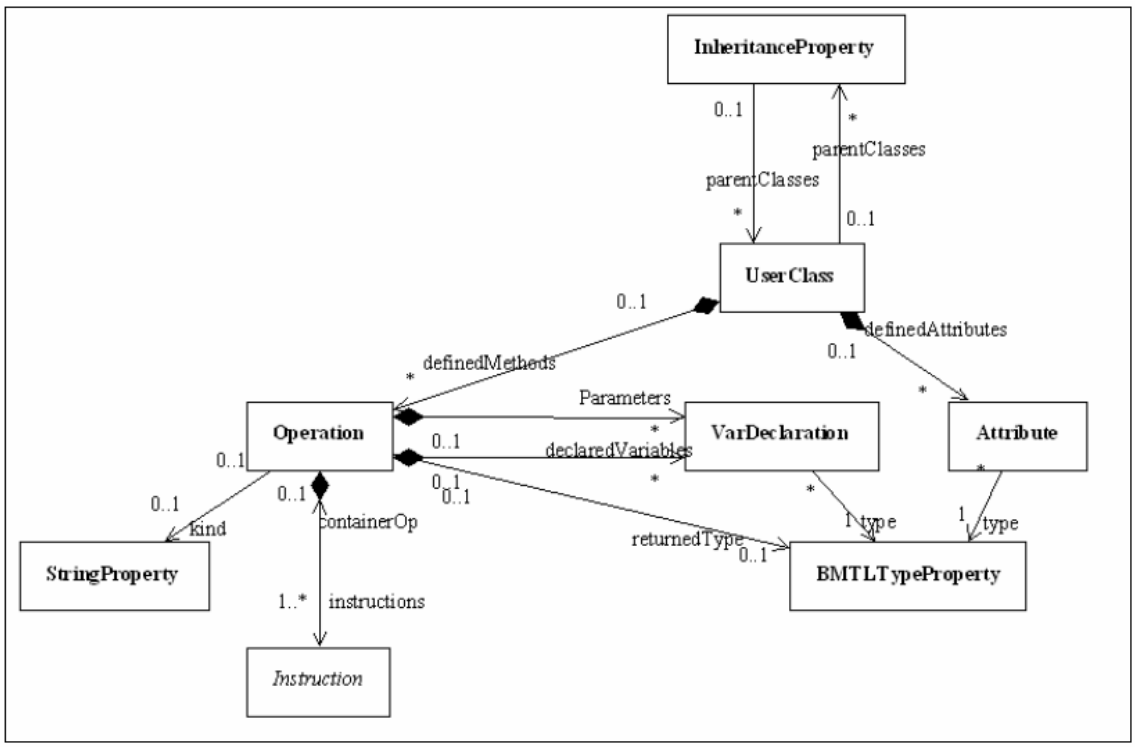

Figura 9 – Meta-modelo BasicMTL – Classes de Usuário e Operações [MTL05].

Na BasicMTL uma *UserClass* tem o mesmo conceito de *Class* em MOF. É um tipo que possui objetos como suas instâncias. Uma *UserClass* possui atributos e operações, é permitida herança múltipla. Os conceitos do repositório "*class*" e "*UserClass*" são similares e podem ser ligados. Uma *operation* pertence a uma *UserClass* e pode ser chamada no contexto

de objetos que são instâncias da *UserClass*. É possível chamar uma operação em qualquer objeto que está diretamente ou indiretamente em uma instância da *UserClass*.

A BasicMTL possui uma biblioteca padrão, esta biblioteca define principalmente os tipos primitivos, eles foram derivados dos tipos da OCL 2.0. Os tipos base incluem: OclAny, ModelElement, BMTLObject, OCLType, BMTLLibrary, Real, Integer, String e Boolean. Os tipos relacionados com Collection são: Iterator, Collection, Set, OrderedSet, Bag e Sequence.

Algumas vantagens obtidas com o uso da linguagem BasicMTL são:

- Melhorar a produtividade da engenharia de software, pois a transformação de modelos contribui para automatizar a construção dos produtos de trabalho de software.
- Aumentar o conhecimento sobre as transformações de modelos – utilização de padronização para garantir a portabilidade e a durabilidade.

4.3 Conclusões

Neste capítulo foram apresentados os conceitos de Action Semantics e BasicMTL, duas abordagens de transformação de modelos que utilizam os conceitos de meta-modelo e restrições em OCL. No Capítulo 5 será apresentada a forma utilizada para introduzir os conceitos de métodos formais na linguagem RDL. Será apresentado o meta-modelo da linguagem, os comandos disponibilizados por ela, e as restrições aplicáveis a cada comando.

5 FORMALIZAÇÃO

Para viabilizar a inclusão de métodos formais na linguagem RDL, foi definido o uso da abordagem de meta-modelos. O processo de verificação ocorre através de três etapas:

- **Modelagem:** esta etapa consiste em construir um modelo formal da linguagem (o meta-modelo RDL), fazendo uso da linguagem UML, e a partir dele obter os comportamentos possíveis dos modelos criados a partir meta-modelo definido.
- **Especificação:** esta etapa consiste em especificar as propriedades comportamentais desejáveis. Um comportamento que se deseja do modelo pode ser descrito formalmente através de restrições escritas em OCL.
- **Verificação:** esta etapa consiste em verificar se as especificações escritas são satisfeitas pelos modelos criados. Esta etapa é feita de forma automática por uma ferramenta.

Na Seção 5.1 é apresentado o diagrama de classes do meta-modelo RDL, em seguida será descrito o memory model da linguagem e as restrições aplicáveis a cada comando.

5.1 Meta-modelo de RDL

A linguagem RDL fornece sintaxe e semântica para o processo que manipula modelos como os presentes no meta-modelo da UML. O meta-modelo da UML consiste em diagramas de classe UML. A consistência sintática garante a conformidade dos modelos com o meta-modelo da linguagem de modelagem [OMG99]. No caso da UML, isto significa que o modelo UML definido pelo usuário deve estar conforme o seu meta-modelo.

A linguagem RDL atualmente está formalizada através de sua BNF (Anexo I), onde são definidas as restrições sintáticas da linguagem. Na Tabela 2 está representada uma parte da BNF da RDL. Está definido que:

- COOKBOOK deve ter um nome, zero ou mais recipes e um main.
- IP_RECIPE tem um comentário (opcional), deve ter um nome e um corpo.
- IP_RECIPE_BODY pode ter vários comandos.

Tabela 2 – BNF da linguagem RDL.

COOKBOOK ::=	**Cookbook** NAME IP_RECIPE* IP_MAIN **end_cookbook;**
IP_RECIPE ::=	[COMMENT_EXP] **recipe** IP_NAME; IP_RECIPE_BODY **end_recipe**;
IP_RECIPE_BODY ::=	IP_CMD*

A partir da BNF, foi derivado o meta-modelo de RDL representado através de um diagrama de classes UML, sendo esta a primeira contribuição deste trabalho. O meta-modelo possui as mesmas restrições estáticas da BNF.

O meta-modelo RDL é construído sobre o meta-modelo de UML. As meta-classes *Class*, *Attribute* e *Method* definidas pelo meta-modelo de UML – Core Package da UML [UML99] – são reutilizadas e estendidas no meta-modelo RDL. A partir do meta-modelo de RDL, pode-se definir um conjunto de regras de boa formação para garantir a conformidade sintática entre os modelos e o meta-modelo. Na Figura 10 está representado o meta-modelo de RDL. Este meta-modelo contém uma classe com um *Cookbook* que contém um *Script* explicando como deve ser executada a instanciação do framework. O *Script* contém um conjunto de *Recipes*. Estas *Recipes* contêm o código de instanciação. Um *Script* pode conter várias *Recipes*, *Pattern* e *Command*, maiores detalhes sobre estas classes podem ser encontrados em [OLI01]. A classe *ReusableAsset* representa o artefato reutilizável, um conjunto de documentos desenvolvidos pelo desenvolvedor do framework e requeridos para a instanciação do framework. A classe *InputModel* representa o modelo de entrada (um diagrama de classes UML (proveniente do framework)). A classe *OutputModel* representa o modelo de saída (o diagrama de classes da aplicação final).

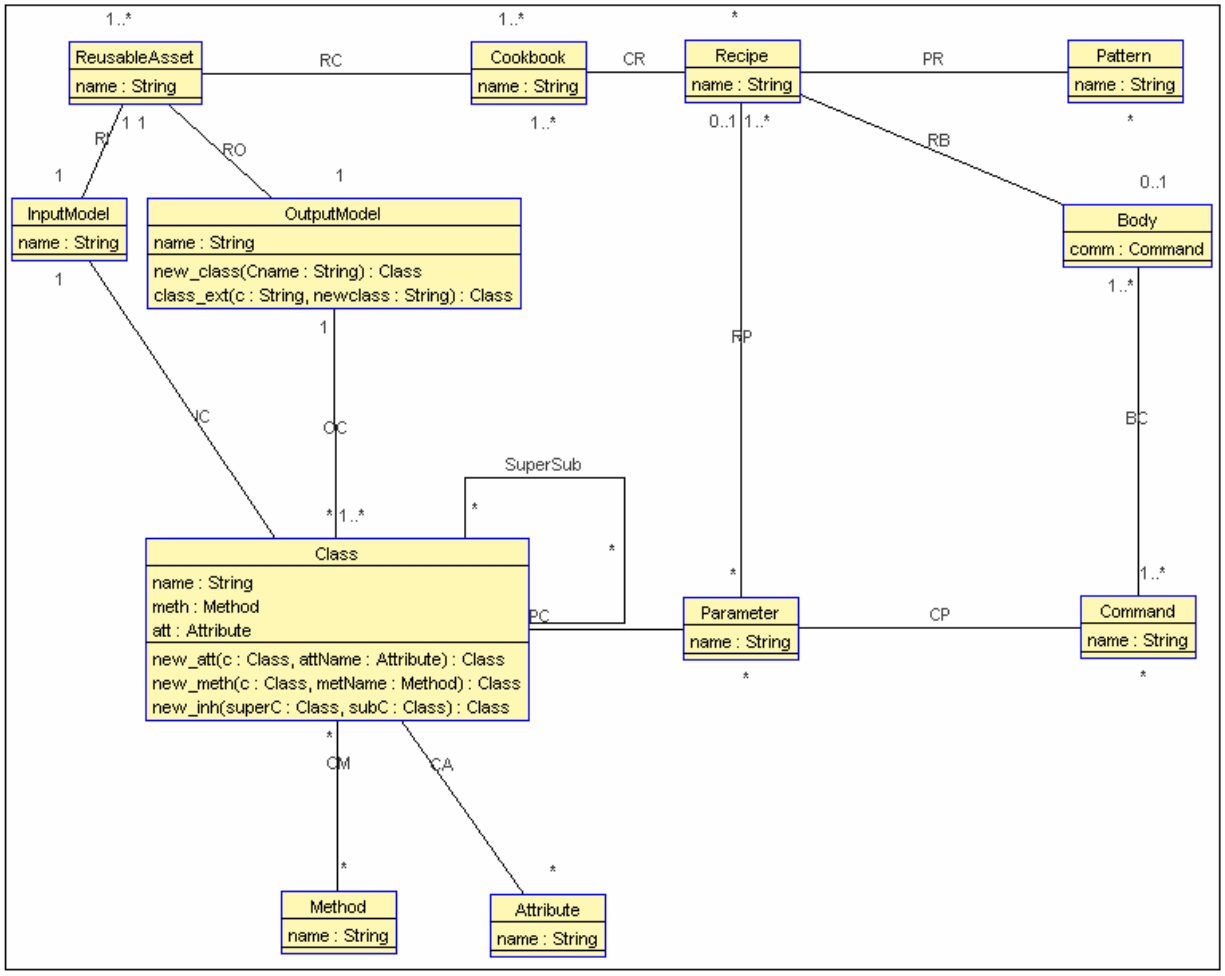

Figura 10 – Meta-modelo RDL.

A semântica de uma linguagem define que construções dentro de uma linguagem são bem formadas e tipadas. As regras de semântica são aplicáveis somente sobre modelos que são sintaticamente bem construídos.

Basicamente são definidas várias restrições sobre as construções sintáticas e o relacionamento entre elas. As restrições serão definidas e explicadas mais adiante.

5.2 Memory Model

RDL possui três arquivos de referência: *(i)* um modelo de entrada representado por um diagrama de classes UML; *(ii)* um modelo RDL (script) e *(iii)* um modelo de saída. Estes três arquivos compõem o Memory Model.

O modelo de entrada possui um conjunto de classes previamente definidas no

framework para representação da atividade de instanciação. Na atividade de reuso, as características necessárias de uma nova aplicação são inseridas em um script RDL, e a partir deste script é gerado o modelo de saída.

É importante salientar a existência de restrições que só podem ser verificadas em tempo de execução. Um exemplo é na criação de uma nova classe. Quando se cria uma nova classe deve-se verificar no modelo de entrada e no modelo de saída se já existe alguma classe com o nome da classe que está sendo criada, pois não é permitida a existência de duas classes com o mesmo nome no modelo de saída. Outro exemplo é quando se faz uma extensão de classes. Deve-se verificar se a classe que está sendo estendida existe no modelo de entrada, em seguida deve-se copiá-la para o modelo de saída e então fazer a extensão.

Na Figura 11 o diagrama de classes do modelo de entrada possui três classes definidas: a classe A, classe B e a classe X, e existe uma relação entre a classe A e a classe X. O script RDL foi definido com as instruções de inclusão de uma nova classe (D), indicação de uma relação de herança entre as classes A e D e inclusão de um novo método (M) na classe D. E na seqüência está o diagrama de classes do modelo de saída correspondente ao script.

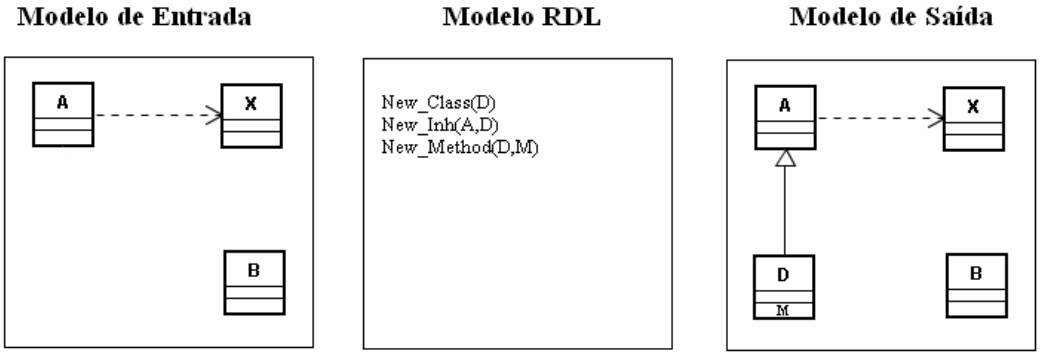

Figura 11 – Modelos.

A linguagem RDL deve garantir que o modelo de entrada, o script do modelo RDL e o modelo de saída estão bem formados, ou seja, que os três modelos estão consistentes e a forma que ela irá realizar isto é a partir da definição de restrições aplicáveis aos modelos. Estas restrições serão definidas na seção 5.3.

5.3 Comandos RDL e Restrições

Na definição das restrições para os modelos deve-se garantir que o modelo de

entrada, o script RDL e o modelo de saída estão bem formados, ou seja, que eles estão consistentes.

RDL possui um conjunto de elementos da linguagem relacionados às tarefas de instanciação. De acordo com o propósito destas tarefas, RDL está dividida em cinco grupos: Tarefas Básicas de Programação, Tarefas Específicas de Instanciação, Tarefas Específicas de Padrões, Tarefas Específicas de Seqüência e Tarefas Diversas. A seguir são apresentados os comandos RDL, as sintaxes e as restrições aplicáveis a estes comandos.

As restrições que garantem que o script está bem formado estruturalmente são relacionadas à estrutura sintática do script, são chamadas de restrições estáticas, por exemplo, um Cookbook deve ter um nome, zero ou mais recipes e uma recipe main; uma recipe deve ter um nome; uma classe deve ter um nome. A restrição que garante que um Cookbook possui uma Recipe chamada Main é a seguinte:

```
context Cookbook inv CookbookHasMain:
self.recipe->exists(r|r.name='Main')
```

As restrições que são definidas em tempo de execução são chamadas de restrições dinâmicas, elas só podem ser verificadas durante a execução do script RDL, elas verificam se o programa executa corretamente. Restrições dinâmicas se aplicam, por exemplo, na utilização do comando **`new_class`**. Quando uma nova classe é criada, deve-se verificar no modelo de entrada e de saída se esta classe já existe, para evitar a ocorrência de duas classes com o mesmo nome e esta verificação só pode ser feita durante a execução do script RDL.

5.3.1 Tarefas Básicas de Programação

Tarefas Básicas de Programação representam o projeto e atividades de manipulação de código. Os comandos básicos fornecem facilidades de baixo nível para manipular os elementos do projeto do framework. Por exemplo, novas classes, métodos ou atributos podem ser criados e adicionados aos modelos dos diagramas de classe UML.

5.3.1.1 New_Class

É um comando para criação de classes. Este comando indica a ação de criar uma classe chamada *cName* no projeto.

Sintaxe: *class* **new_class**(*cName*).

Código:

```
COOBOOK Example
  RECIPE main
     //  adding a new class
     NEW_CLASS (Class2);
     //…
  END_RECIPE;
END_COOKBOOK
```

Restrição: Quando se cria uma nova classe deve-se verificar no modelo de entrada e de saída se já existe alguma classe com o mesmo nome da classe que está sendo criada, pois não são permitidas duas classes com o mesmo nome no mesmo container. A restrição em OCL que garante que não existem duas classes com o mesmo nome é a seguinte:

```
context Classe inv ClassNameUnique: Classe.allInstances-> forAll
(other|self.name = other.name implies self=other)
```

A pré-condição aplicada ao modelo de saída, que garante que o nome da classe que está sendo criada é diferente do nome da classe, está a seguir:

```
context OutputModel:: new_class(Cname: Classe): Classe
  pre: (self.name <> Cname.name)
```

5.3.1.2 New_Method

É um comando para a criação de métodos. Este comando indica a criação de um método chamado *metName* na classe *c*.

Sintaxe: *method* **new_method** (*c*, *metName*).

Código:

```
COOBOOK Example
  RECIPE main
     //  adding a new method
     NEW_METHOD (Class1,operation1);
     //
  END_RECIPE;
END_COOKBOOK
```

Restrição: Quando se cria um novo método deve-se verificar se já existe na classe um método com o mesmo nome do método que está sendo criado, pois não são permitidos dois métodos com o mesmo nome na mesma classe. A restrição em OCL que garante que não existem dois métodos com o mesmo nome é a seguinte:

```
context Classe inv MethodNameUnique: Method.allInstances-> forAll
```

```
(other|self.name = other.name implies self=other)
```

5.3.1.3 New_Attribute

Comando para criação de um atributo. Este comando indica a criação de um atributo chamado *attName* na classe *c*.

Sintaxe: *attribute* **new_attribute** (*c*, *attName*).

Código:

```
COOBOOK Example
  RECIPE main
     //  adding a new attribute
     NEW_ATTRIBUTE (Class1,attribute1);
     //
  END_RECIPE;
END_COOKBOOK
```

Restrição: Quando se adiciona um novo atributo a uma classe deve-se verificar a existência de um atributo com o mesmo nome na classe, pois não são permitidos dois atributos com o mesmo nome na mesma classe. A restrição em OCL que garante que não existem dois atributos com o mesmo nome é a seguinte:

```
context Classe inv AttributeNameUnique: Attribute.allInstances->

forAll (other|self.name = other.name implies self=other)
```

5.3.1.4 New_Inheritance

Este é um comando para definição de herança. Ele indica o estabelecimento de um relacionamento de herança entre as classes *SuperC* e *SubC*.

Sintaxe: *void* **new_inheritance** (*superC*, *subC*).

Código:

```
COOBOOK Example
  RECIPE main
     //  defining an inheritance
     NEW_INHERITANCE(Class1,Class2);
     //
  END_RECIPE;
END_COOKBOOK
```

Restrição: Quando se define uma relação de herança entre duas classes deve-se certificar de que já não existe uma relação de herança entre estas duas classes. A pré-condição aplicada a este comando é a seguinte:

```
context Classe:: new_inh(superC: Classe, subC: Classe): Classe
  pre: (self.name <> superC.name) and (self.name <> subC.name) and
     (subC.name < superC.name)
```

5.3.2 Tarefas Específicas de Instanciação

Tarefas Específicas de Instanciação geralmente combinam algumas tarefas básicas encontradas no grupo das Tarefas Básicas de Programação. Os comandos de instanciação aumentam o nível de abstração através da combinação de comandos básicos em tarefas simples. Basicamente, os comandos de instanciação RDL representam atividades de reuso orientados a objetos como extensão de classes, extensão de métodos, atribuição de valores a um atributo de uma classe.

5.3.2.1 Element_Choice

O comando escolha de um elemento indica a ação de escolher se o elemento *el* estará presente no projeto da aplicação final. Ele retorna *true* se o elemento for escolhido e *false* se não for. Esta ação depende de informação somente disponível na hora da instanciação, pois necessita de feedback do usuário.

Sintaxe: *boolean* **element_choice** (*el*)

Código:

```
COOBOOK Example
  RECIPE main
     //  choosing an element
     ELEMENT_CHOICE (C1.aComponent);
     //
  END_RECIPE;
END_COOKBOOK
```

Restrição: Se o elemento *el* for escolhido, deve-se garantir que ele estará no projeto da aplicação final e se ele não for escolhido, deve-se certificar de que não existem elementos dependentes deste elemento que tornem a sua escolha obrigatória.

5.3.2.2 Class_Extension

Este comando indica a ação de estender uma classe *C* com uma nova classe chamada *cName*. Esta ação executa os comandos de criação de classe e definição de herança.

Sintaxe: *class* **class_extension** (*C*, *cName*)

Código:

```
COOBOOK Example
  RECIPE main
     //  extending a class1
     CLASS_EXTENSION (Class1,NewClass);
     //
  END_RECIPE;
END_COOKBOOK
```

Restrição: Nesta extensão de classe deve-se certificar de que na criação da nova classe não existe uma classe com o mesmo nome da classe que está sendo criada e deve-se garantir que não existe nenhuma relação de herança entre elas. Esta ação executa ambos os comandos Criação de Classe e Definição de Herança, já citados anteriormente.

5.3.2.3 Select_Class_Extension

Este comando indica a ação de estender uma classe C através da seleção de uma de suas subclasses concretas e retorna a classe selecionada. Esta ação lista todas as subclasses concretas da classe C, assim o reutilizador pode escolher uma. É importante ressaltar que as classes não selecionadas serão removidas do projeto da aplicação se, e somente se, não houver referências a elas.

Sintaxe: *class* **select_class_extension**(*C*)

Código:

```
COOBOOK Example
  RECIPE main
     //  extending a class1 through selection
     SELECT_CLASS_EXTENSION (class1);
     //
  END_RECIPE;
END_COOKBOOK
```

Restrição: Nesta operação deve-se certificar de que a classe selecionada existe no modelo de entrada e deve-se garantir que esta classe estará presente no modelo de saída.

5.3.2.4 Method_Extension

Este comando indica a redefinição de um método *metName* na subclasse *subC*,onde *metName* é um método declarado na *superC* e *superC* é a superclasse de *subC*. Esta ação executa o comando criação de método e checa o relacionamento de herança entre duas classes.

Sintaxe: *void* **method_extension**(*superC*, *metName*, *subC*)

Código:

```
COOBOOK Example
  RECIPE main
     //  redefining a method
     METHOD_EXTENSION (Class1, metM, Class2);
     //
  END_RECIPE;
END_COOKBOOK
```

Restrição: Nesta redefinição deve-se certificar que a superclasse e a subclasse estão presentes no modelo e se existe uma relação de herança entre elas. Esta ação executa o comando Criação de Método e verifica o relacionamento de herança entre as duas classes.

5.3.3 Tarefas Específicas de Padrões

Tarefas Específicas de Padrões suportam a reunião de vários comandos RDL em uma receita especial, o Script Padrão, é desenvolvido para executar uma seqüência de ações de instanciação que são relevantes para o domínio do Framework. Os Scripts Padrões promovem o reuso, por meio de um nome, das atividades RDL em processos de instanciação recorrentes.

5.3.3.1 Pattern_Class_Extension

Este comando indica a ação de estender uma classe *C* através do padrão *patName* utilizando os parâmetros presentes na lista *L*. A extensão da classe Pattern cria no mínimo uma subclasse da classe *C* no projeto final e talvez adiciona outros elementos no projeto especificados no script pattern. A maneira que a lista dos parâmetros é utilizada é totalmente dependente da maneira que o script pattern é implementado. O exemplo a seguir mostra como o Pattern AbstractFactory pode ser especificado e aplicado para estender uma classe.

Sintaxe: *void* **pattern_class_extension**(*C*, *patName*, *L*)

Código:

```
COOBOOK Example
  RECIPE main
     //  pattern class extension    PATTERN_CLASS_EXTENSION( AbsFactory,
AbstractFactory,(AbsFactory ,AbsProduct));
     //
  END_RECIPE;
```

```
END_COOKBOOK


PATTERN AbstractFactory (AbsFactory , AbsProduct)
  // This is the pattern script
  c1 = CLASS_EXTENSION (AbsFactory,?);
    METHOD_EXTENSION(AbsFactory,createProduct,c1);
```

Restrição: No uso de um pattern para estender uma classe, deve-se verificar na Syntax Table[1] se o tipo da variável que está recebendo a extensão da classe é do tipo classe e deve-se aplicar a este comando a mesma restrição aplicada à extensão de classes, deve-se certificar de que na criação da nova classe não existe uma classe com o mesmo nome da classe que está sendo criada e deve-se garantir que não existe nenhuma relação de herança entre elas. Esta ação executa ambos os comandos Criação de Classe e Definição de Herança.

5.3.3.2 Pattern_Method_Extension

Este comando indica a ação de redefinir o método *MetName* na subclasse *subC* através do pattern *patName* usando os parâmetros presentes na lista *L*. A extensão do método Pattern redefine o método *metName* na subclasse *subC* e talvez adicione outros elementos no projeto especificados no script pattern. A maneira que a lista dos parâmetros é utilizada é totalmente dependente da maneira que o script pattern é implementado.

Sintaxe: *void* **pattern_method_extension** (*superC*, *metName*, *subC*, *patName*, *list*)

Código:

```
COOBOOK Example
  RECIPE main
     //  pattern method extension    PATTERN_METHOD_EXTENSION( Subject,
request, RealSubject(…));
     //
  END_RECIPE;
END_COOKBOOK

Pattern Script:
PATTERN Strategy (…)
  // This is the pattern script
  c1 = CLASS_EXTENSION (Subject,?);
m1 = METHOD_EXTENSION(Subject,request,c1);
  NEW_ATTRIBUTE (c1,realSubject);
  ADD_CODE(c1,m1,"//add code …" );

END_PATTERN
```

Restrição: No uso de um pattern para estender um método, deve-se verificar na Syntax Table se o tipo da variável que está recebendo a extensão da classe é do tipo classe e deve-se aplicar

[1] Syntax Table é uma classe no modelo de execução que guarda os tipos de dados dos atributos. Deve existir uma Sintax Table no modelo de entrada e no modelo de saída.

a este comando a mesma restrição aplicada à extensão de classes e métodos, deve-se certificar que a superclasse e a subclasse estão presentes no modelo e se existe uma relação de herança entre elas. Esta ação executa o comando Criação de Método e verifica o relacionamento de herança entre as duas classes.

5.3.4 Tarefas Específicas de Seqüência

As Tarefas Específicas de Seqüência descrevem a maneira que os comandos RDL são combinados no fluxo de execução. RDL também permite a especificação de dependências de estado indicando se um elemento do projeto é requerido na aplicação final.

5.3.4.1 And (#)

O comando *and* indica que ambos os comandos *cmd1* e *cmd2* devem ser executados nesta ordem. A seqüência *and* é expressa naturalmente em RDL quando se coloca os comandos linha após linha.

Sintaxe: *cmd1* **#** *cmd2*

Código:

```
COOBOOK Example
  RECIPE main
     // and
     CLASS_EXTENSION (Class1,ClassA) # CLASS_EXTENSION (Class2,ClassB);
  END_RECIPE;
END_COOKBOOK
```

Restrição: Quando se utiliza o comando *and* deve-se garantir que o comando que está antes do *and* e o comando que está depois foram executados e o resultado desta operação aparece no modelo de saída. Por exemplo, no modelo de saída do trecho de código acima deve aparecer a classe1 e a classeA com uma relação de herança e a classe2 com uma relação de herança com a classeB.

5.3.4.2 Or (o)

O comando *or* indica que somente um comando deve ser executado. A seqüência *or* pergunta ao reutilizador do framework qual opção ele irá executar.

Sintaxe: *cmd1* **o** *cmd2*

Código:

```
COOBOOK Example
  RECIPE main
     // or
     CLASS_EXTENSION (Class1,ClassA) o CLASS_EXTENSION (Class2,ClassB);
  END_RECIPE;
END_COOKBOOK
```

Restrição: Quando se utiliza o comando *or* deve-se garantir que um dos comandos será executado, ou o comando que está antes do *or*, ou o comando que está depois e deve-se garantir que o resultado desta operação aparece no modelo de saída. Por exemplo, no trecho de código acima, supondo que foi escolhida a extensão da classe1, no modelo de saída deve aparecer a classe1 e a classeA com uma relação de herança e a classe2.

5.3.4.3 Parallel Execution

Este comando indica que o *cmd1* e o *cmd2* podem ser executados concorrentemente. A seqüência *//* é escolhida para especificar processos de instanciação de frameworks que podem ser manipulados por equipes diferentes.

Sintaxe: *cmd1* **o** *cmd2*

Código:

```
COOBOOK Example
  RECIPE main
     // parallel
     CLASS_EXTENSION (Class1,ClassA) // CLASS_EXTENSION (Class2,ClassB);
  END_RECIPE;
END_COOKBOOK
```

Restrição: Na utilização deste comando a ordem de execução não é relevante, somente é necessário garantir que o resultado da execução dos comandos aparece na aplicação final.

Quando são utilizados os comandos das Tarefas Específicas de Seqüência, as operações são definidas em tempo de execução e o contexto dos operadores de fluxo é muito amplo, por exemplo, podem ser classes, métodos ou atributos. Nestes casos a responsabilidade pela verificação da execução dos comandos fica a cargo do projetista do framework.

5.3.5 Tarefas Diversas

As Tarefas Diversas são aquelas que não se adaptam àquelas agrupadas previamente.

5.3.5.1 Repetition

Este comando indica que os *cmds* são repetidos um número de vezes. O número de iterações é definido pelo reutilizador quando ele está utilizando o ambiente de execução RDL.

Sintaxe: **loop** *cmd1* **end_loop**

Código:

```
COOBOOK Example
  RECIPE main
     // loop
    LOOP
     CLASS_EXTENSION (Class1, ?);
    END_LOOP;
  END_RECIPE;
END_COOKBOOK
```

Restrição: Não há nenhuma restrição para a execução deste comando.

5.3.5.2 Assignment

Este comando indica que o resultado da execução do comando *cmd1* é armazenado em *var*.

Sintaxe: *var* = *cmd1*

Código:

```
COOBOOK Example
  RECIPE main
     // assignment
     c1 = CLASS_EXTENSION (Class1,?);
     NEW_METHOD(c1,M1);

  END_RECIPE;
END_COOKBOOK
```

Restrição: Quando se executa o comando *assignment* deve-se verificar na Syntax Table se o tipo da variável que está recebendo o valor é o mesmo tipo do valor que está sendo atribuído a ela.

5.3.5.3 Reuser Interaction

Este comando indica que o ambiente de execução RDL deve interagir com o reutilizador para obter informações específicas do domínio.

Sintaxe: ?

Código:

```
COOBOOK Example
  RECIPE main
     // reuser interaction
     CLASS_EXTENSION (Class1,?);
  END_RECIPE;
END_COOKBOOK
```

Restrição: Não há nenhuma restrição definida para este comando.

5.4 Conclusões

Este capítulo apresentou o meta-modelo da RDL que é um modelo que representa a estrutura de um conjunto particular de modelos e é usado para descrever os projetos expressos em RDL. Foram apresentados os Modelos de Entrada, script RDL e o Modelo de Saída utilizados pela linguagem RDL.

A especificação da RDL descreve a sintaxe abstrata da linguagem através de um subconjunto das notações de modelagem estática e através da Linguagem de Restrições de Objetos (OCL). Foram apresentados neste capítulo os comandos da linguagem RDL. Foram definidas as restrições aplicáveis a todos os comandos, e a forma que estas restrições são utilizadas na linguagem OCL. É importante ressaltar que nem todas as restrições definidas em linguagem natural foram mapeadas para OCL. Isto porque quando são utilizados os comandos das Tarefas Específicas de Padrões e Tarefas Específicas de Seqüência, por exemplo, as operações são definidas em tempo de execução e o contexto dos operadores de fluxo é muito amplo, por exemplo, podem ser classes, métodos ou atributos. Nestes casos a responsabilidade pela verificação da execução dos comandos fica a cargo do reutilizador do framework.

No Capítulo 6 será apresentada a Ferramenta USE, a ferramenta utilizada para mapear as restrições definidas em linguagem natural para a linguagem OCL.

6 TESTES

Este capítulo apresenta a segunda contribuição deste trabalho: os testes da corretude das restrições definidas no Capítulo 5 utilizando a ferramenta USE.

6.1 Meta-modelo RDL

O meta-modelo da linguagem RDL é formado por uma sintaxe abstrata (modelo) e por um conjunto de restrições, como pode ser visualizado na Figura 12. A sintaxe abstrata descreve a sintaxe dos elementos de RDL utilizando um diagrama de classes, ela formaliza aspectos relevantes da linguagem com o objetivo de garantir a corretude, completude, consistência, concisão e clareza dos modelos. Todo este formalismo durante a especificação, permite a redução de ambigüidade e inconsistência. O conjunto de restrições descreve restrições de formação utilizando OCL.

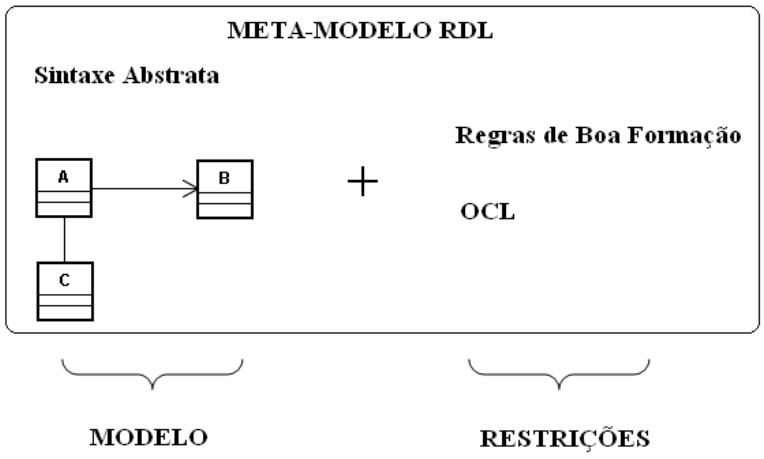

Figura 12 – Meta-modelo RDL e Restrições

A ferramenta USE foi utilizada para testar os modelos criados com a linguagem RDL, foram utilizados o modelo de entrada, o script RDL descrevendo as instanciações de classes e operações a serem realizadas e o meta-modelo RDL definido anteriormente para testar os modelos criados, conforme a Figura 13.

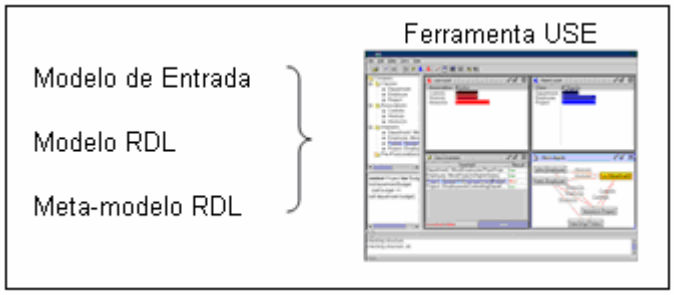

Figura 13 – Testes

6.2 Ferramenta USE

A ferramenta USE foi desenvolvida pela Universidade de Bremen para permitir a especificação de sistemas de informação usando um subconjunto da linguagem UML e permitir demonstrar restrições sobre esses modelos através de expressões OCL, que podem ser verificadas com o avaliador de expressões OCL que a ferramenta inclui. Além disso, permite que o sistema especificado seja populado e que se possa obter informação detalhada sobre estes modelos, novamente recorrendo à OCL [USE06].

Outra característica interessante desta ferramenta é a sua capacidade para permitir carregar especificações de modelos. Em particular, pode-se carregar o meta-modelo da UML 2.0, popular o meta-modelo com meta-objetos e extrair informação sobre esses meta-objetos. Na prática, isso significa que se pode recolher métricas formalizadas em OCL, navegando nos meta-objetos criados, bastando para tal que os meta-objetos representem a infra-estrutura de componentes que se pretende avaliar.

6.2.1 Utilização da ferramenta USE

Uma especificação em USE contém uma descrição textual (classes, associações, atributos, operações e restrições) de um diagrama de classes UML. Esta descrição textual é própria da ferramenta. É possível com a USE animar um modelo e assim validá-lo de acordo com os requisitos de um dado sistema. Tal é feito através do uso de restrições especificadas em OCL. A Figura 14 ilustra a funcionalidade desta ferramenta [USE06].

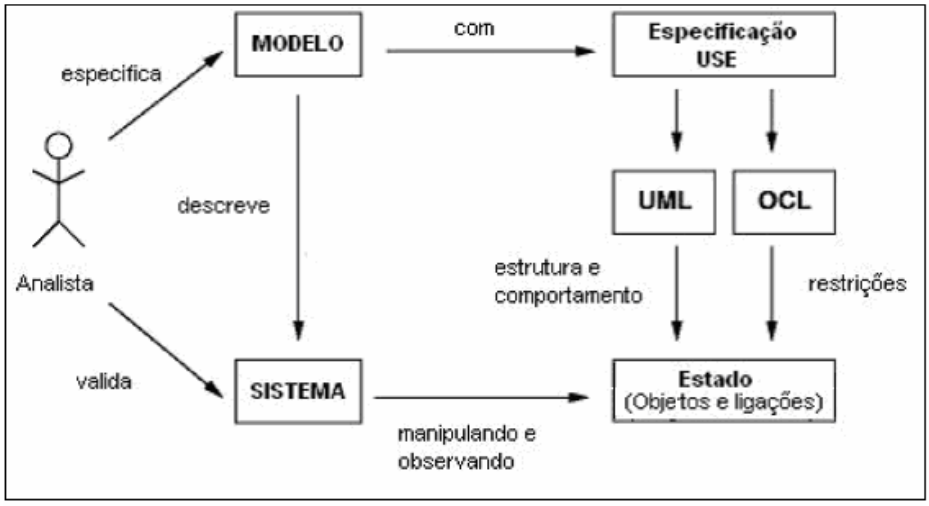

Figura 14 – Funcionalidade da ferramenta USE

A ferramenta USE pode ser utilizada de duas formas, através de uma interface gráfica ou através de uma linha de comando. A ferramenta inclui um carregador de modelos. Dado um diagrama de classes, a ferramenta permite popular esse diagrama com objetos apropriados, sendo depois possível avaliar expressões OCL sobre o modelo populado. Além de se poder verificar com o avaliador de expressões OCL se o estado do modelo carregado obedece às restrições impostas em OCL, é também possível recolher a informação relevante e detalhada sobre o estado do modelo. Estas características são essenciais para o nosso trabalho: podemos carregar o meta-modelo da RDL e popular esse meta-modelo com os elementos correspondentes à infra-estrutura de componentes que pretendemos avaliar.

A definição dos meta-modelos é feita em um arquivo texto, onde são descritas todas as classes do meta-modelo, seus atributos, associações e restrições. O trecho a seguir mostra partes da definição de um meta-modelo de uma locadora de veículos – este meta-modelo está disponível nos arquivos de exemplos da ferramenta – podemos observar a definição de uma classe, uma associação e uma restrição.

```
model CarRental2

--  datatypes

--  enum CarGroupKind { compact, intermediate, luxury }
--  enum Sex { male, female }

-- datatype Date "org.tzi.use.uml.ocl.ext.Date"
-- operations
--   static Date(year : Integer, month : Integer, day : Integer) : Date
--   static now() : Date
--   < (date2 : Date) : Boolean
--   > (date2 : Date) : Boolean
--   = (date2 : Date) : Boolean
--   <> (date2 : Date) : Boolean
--   after(when :Date) : Boolean
--   before(when : Date) : Boolean
--   day() : Integer
-- end
```

```
-- classes                                          DEFINIÇÃO DAS CLASSES

abstract class Person
attributes
  firstname : String
  lastname : String
  age : Integer
  isMarried : Boolean
  email : Set(String)
operations
  -- produce a full name, e.g. 'Mr. Frank Black'
  fullname(prefix : String) : String =
    prefix.concat(' ').concat(firstname).concat(' ').concat(lastname)
end

-- Associations                                     DEFINIÇÃO DAS ASSOCIAÇÕES

association Management between
  Employee[1] role manager
  Branch[0..1] role managedBranch
end

-- Constraints                                      DEFINIÇÃO DAS RESTRIÇÕES

Constraints

-- The association Quality is anti-reflexive and anti-symmetric
-- context q1 : Quality inv:
--   not Quality.allInstances->exists(q2 |
--     q1.lower = q2.higher and q1.higher = q2.lower)


context Person

-- The age attribute of persons is greater than zero
inv Person1:
  age > 0
```

Após a definição textual do meta-modelo, o arquivo correspondente é carregado para possibilitar a visualização do diagrama de classes UML, conforme a Figura 15. Para popular o meta-modelo com os objetos, deve-se digitar os comandos – e na seqüência salvar este arquivo .cmd. Quando é feita alguma alteração no modelo de classes, deve-se abrir novamente o arquivo do meta-modelo, e então executar o arquivo de comandos. A ferramenta também possibilita a visualização do diagrama de objetos.

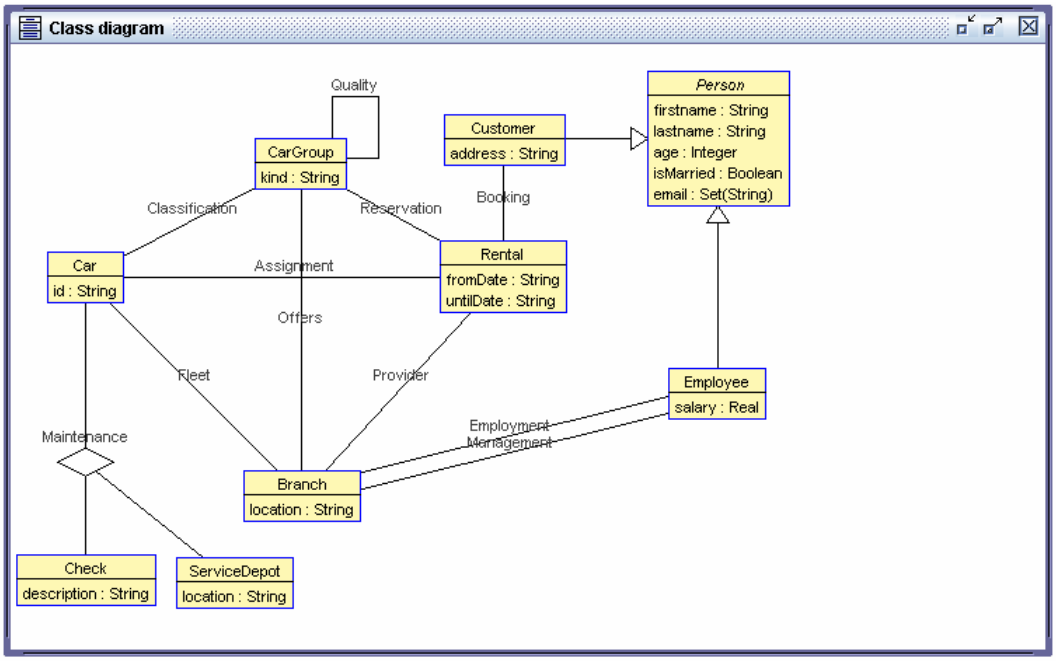

Figura 15 – Meta-modelo Locadora de Veículos [arquivos de exemplos da ferramenta USE]

6.3 Testes do Meta-modelo RDL

O meta-modelo da RDL foi descrito de forma textual – a descrição textual completa do meta-modelo está no Anexo II. O meta-modelo possui 12 classes, 15 associações, 12 invariantes, 5 operações e 4 pré/pós-condições. A Figura 16 mostra a forma que a ferramenta USE disponibiliza as informações referentes ao meta-modelo. No centro está representado o diagrama de classes do meta-modelo, são apresentados os atributos e métodos de cada classe e as multiplicidades das associações. No frame à esquerda na parte superior são apresentadas as classes do meta-modelo, suas associações, invariantes e pré-/pós condições. Para visualizar detalhes de cada um destes elementos, basta clicar sobre o mesmo, que as informações são disponibilizadas no frame do canto inferior esquerdo. No exemplo da Figura 16, foi selecionada a pré-condição `2 new_att,` para ser visualizada em detalhes.

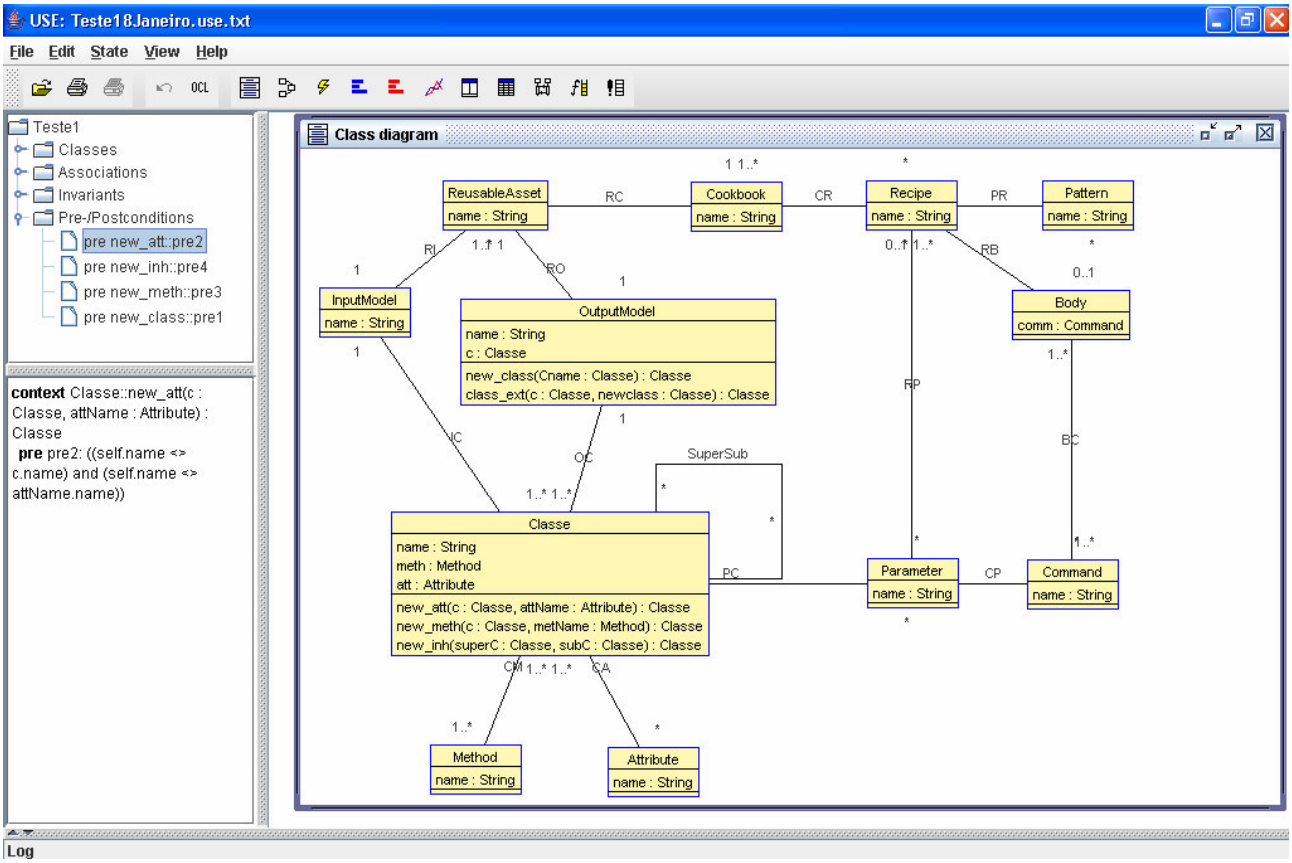

Figura 16 – Meta-modelo na Ferramenta USE

O meta-modelo define a notação da especificação dos modelos através dos relacionamentos e permite que os requisitos sejam interpretados de forma única. O conjunto de restrições define as regras que indicam se os objetos satisfazem a especificação. O script RDL verificado na ferramenta USE foi definido com comandos que não satisfazem algumas restrições definidas no meta-modelo RDL com o objetivo de verificar o resultado da análise destas restrições. Na seqüência é apresentado o primeiro exemplo utilizado para testar as restrições OCL na ferramenta USE, o script completo encontra-se no Anexo III.

6.3.1 Exemplo 1

A Figura 17 ilustra o diagrama de objetos após a inserção das linhas de 1 a 4 do script abaixo:

```
1   !create Y : Recipe
2   !create X : Cookbook
3   !set X.name := 'Cook1'
4   !set Y.name := 'Main'
5   !insert (X,Y) into CR
```

A restrição que garante que um **Cookbook** possui uma **Recipe** chamada Main é a seguinte:

```
context Cookbook inv CookbookHasMain:

self.recipe->exists(r|r.name='Main')
```

Esta restrição não foi satisfeita neste caso e o retorno da ferramenta foi *false.* Na linha 5 esta restrição foi satisfeita quando foi definido o relacionamento entre a classe **Cookbook** e **Recipe**.

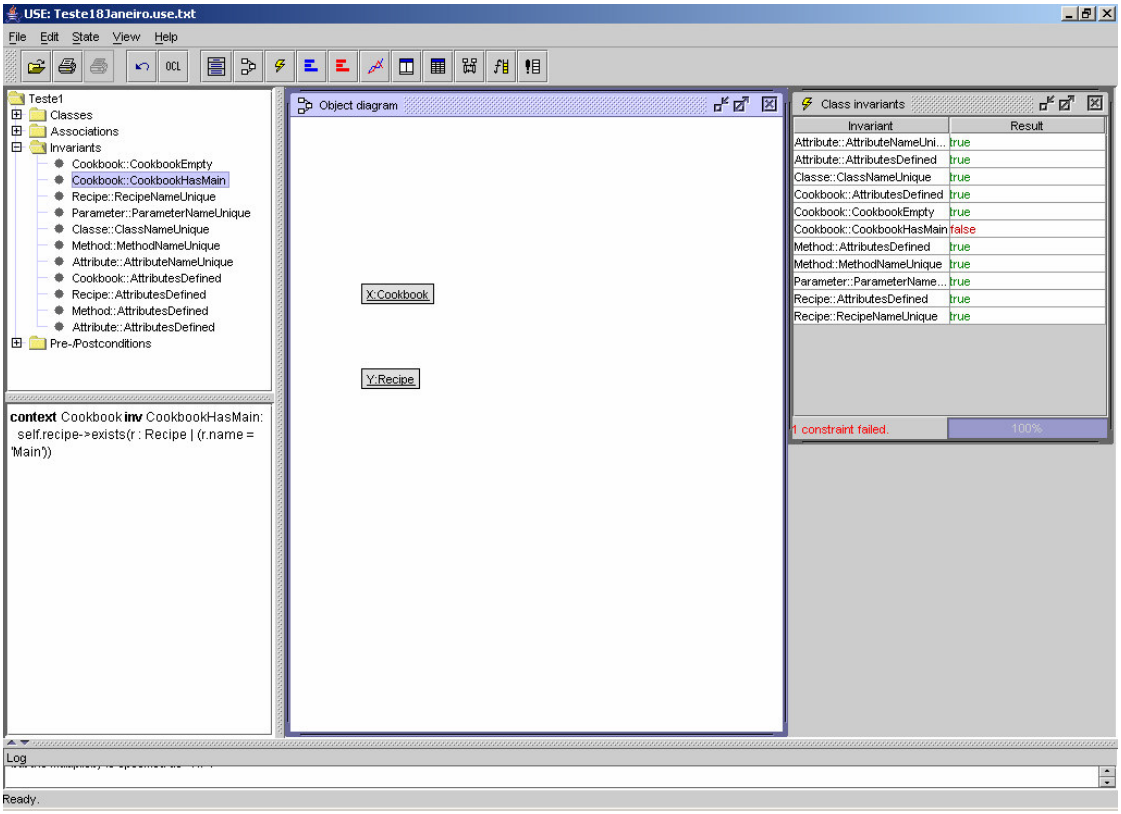

Figura 17 – Restrições OCL

Nas linhas 6 e 7 foi criada uma **Recipe A** e foi atribuído o nome Main para verificar o retorno da restrição que controla a existência de classes com o mesmo nome.

```
6   !create A : Recipe
7   !set A.name := 'Main'
```

A Figura 18 representa o resultado da verificação, a ferramenta USE sinaliza a existência de 2 **Recipes** com o mesmo nome, mas não altera a execução. A restrição que garante a inexistência de 2 **Recipes** com o mesmo nome é:

```
context Recipe inv RecipeNameUnique: Recipe.allInstances-> forAll
(other|self.name = other.name implies self=other)
```

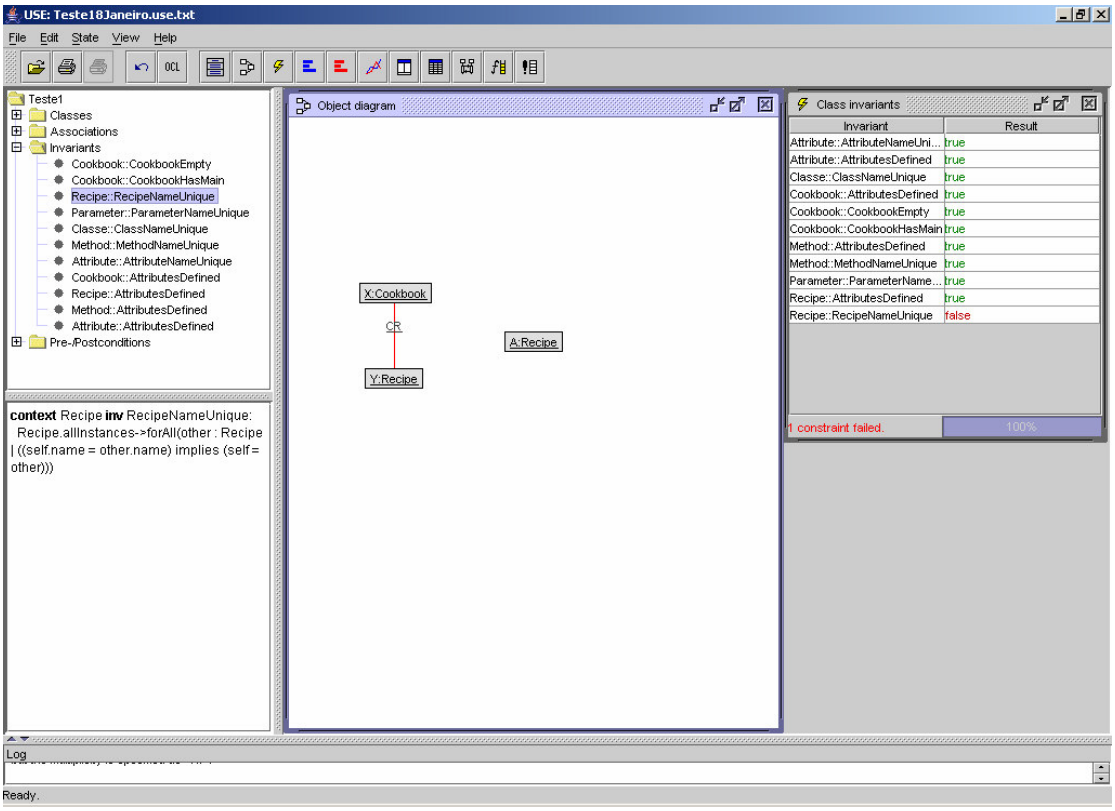

Figura 18 – Restrições OCL

Nas linhas de 41 a 45 foi avaliada a restrição que sinaliza quando existem dois atributos com o mesmo nome. Foi criado um atributo *a* com o nome *C1* e foi criado o atributo *b* com o mesmo nome, a restrição verificou esta inconsistência e retornou *false*.

```
41  !create a : Attribute
42  !set a.name := 'C1'
43  !set Emp.att:= a
44  !create b : Attribute
45  !set b.name := 'C1'
```

Nas linhas de 8 a 34 foi criado um Body, um InputModel, Reusable Asset, Command, Parameter, Classe e OutputModel. Estas classes foram criadas respeitando o meta-modelo RDL, nenhuma restrição foi violada, assim, todas as avaliações resultaram em *true,* como pode ser visto na Figura 19.

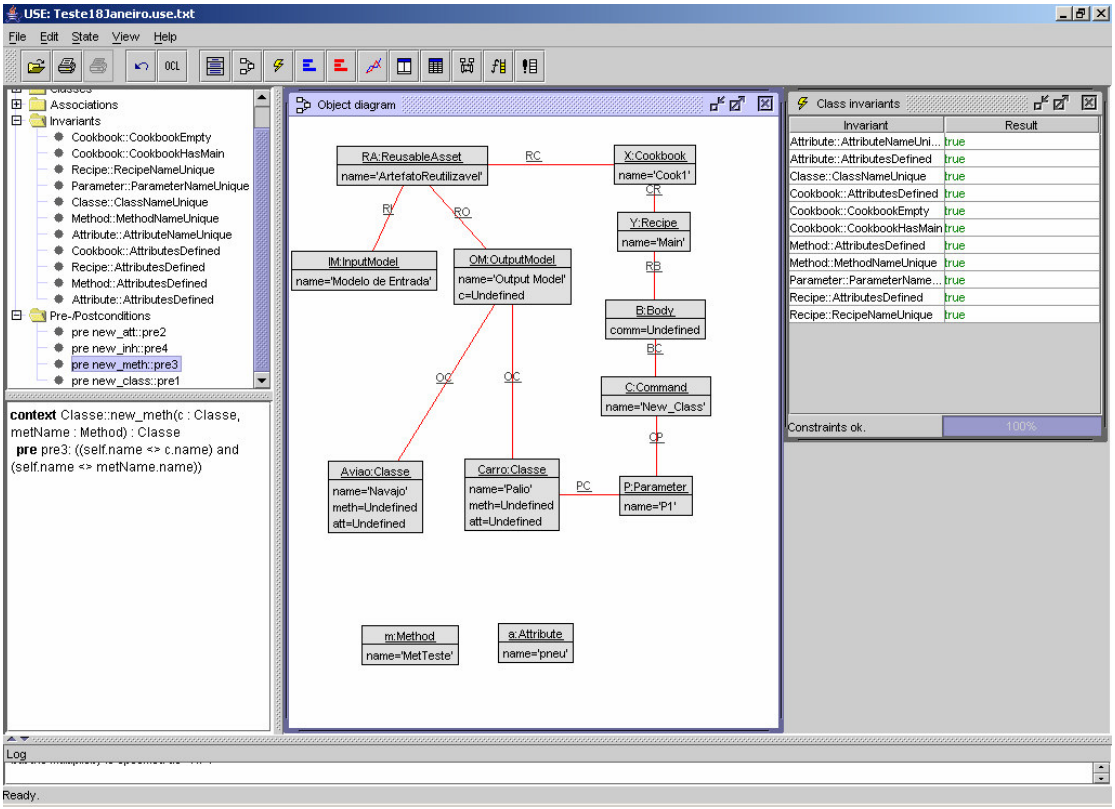

Figura 19 – Restrições OCL

Este primeiro exemplo foi utilizado para verificar se as restrições estáticas estavam avaliando corretamente cada comando. Este objetivo foi alcançado, as avaliações foram realizadas de forma correta. Nas próximas seções, serão apresentados dois scripts RDL que foram definidos pelos autores da linguagem e a sua utilização na ferramenta USE.

6.3.2 Exemplo 2

O script RDL utilizado é apresentado no trecho de código abaixo. Este script define (*i*) a criação de uma subclasse de TestCase, esta nova classe é armazenada na variável testClass; (*ii*) a extensão de dois métodos da superclasse TestCase e (*iii*) a definição de novos atributos e novos métodos para a classe testClass. O parâmetro "?" indica que o nome da nova classe ou do novo método será definido quando o processo de instanciação do framework for executado. O script USE encontra-se no Anexo IV.

```
1.  COOKBOOK JunitFixture
2.   RECIPE main
3.        testClass = CLASS_EXTENSION (TestCase,?);
4.        METHOD_EXTENSION (TestCase, SetUp, testClass);
5.        METHOD_EXTENSION (TestCase, teardown, testClass);
6.        LOOP
```

```
7.              NEW_ATTRIBUTE(testClass, ?);
8.         END_LOOP
9.         LOOP
10.             NEW_METHOD(testClass, ?);
11.        END_LOOP
12.   END_RECIPE
13. END_COOKBOOK
```

Na ferramenta USE foi criado o Cookbook JunitFixture; após a definição do nome do Cookbook, a restrição que verifica se todo o Cookbook possui uma recipe chamada Main retornou *false*, para que a restrição mudasse o seu status para *true* foi definida a Recipe Main. Na linha 3 foi definida uma extensão de classe; como a restrição aplicável a este comando é uma restrição dinâmica, não foi realizada a avaliação desta operação (nos casos que envolvem restrições dinâmicas, a responsabilidade pela avaliação fica para o reutilizador do framework). A mesma situação ocorre nas linhas 4 e 5, onde é realizada uma extensão de métodos, que também é uma restrição dinâmica. Nas linhas de 6 a 11 são definidos, dentro de laços, novos atributos e métodos para a classe testClass, neste caso as restrições OCL que alertam quando existem atributos ou métodos iguais na classe executam corretamente.

6.3.3 Exemplo 3

O script RDL utilizado neste exemplo está no trecho de código a seguir. Este script consiste principalmente de comandos de instanciação. Embora possa parecer óbvio, a criação dos atributos que representam as características do teste deve ser realizada depois da criação da classe que irá conter estes atributos, e a redefinição dos métodos deve ser feita após a criação dos atributos. O script USE encontra-se no Anexo V.

```
1. COOKBOOK JunitChangingResult
2.   RECIPE main
3.        testClass = CLASS_EXTENSION (TestCase, ?);
4.        LOOP
5.             NEW_ATTRIBUTE (TestClass, ?);
6.        END_LOOP
7.        METHOD_EXTENSION (TestCase, setUp, testClass);
8.        METHOD_EXTENSION (TestCase, teardown, testClass);
9.        LOOP
10.            NEW_METHOD (TestClass, ?);
11.       END_LOOP
12.       resultClass =  CLASS_EXTENSION (TestResult, ?);
13.       METHOD_EXTENSION (TestResult, addFailure, resultClass);
14.       METHOD_EXTENSION (TestResult, addError, resultClass);
```

```
15.       METHOD_EXTENSION (TestResult, startTest, resultClass);
16.       METHOD_EXTENSION (TestResult, endTest, resultClass);
17.       METHOD_EXTENSION (TestCase, createResult, resultClass);
18.    END_RECIPE;
19.  END_COOKBOOK
```

Na ferramenta USE foi criado o Cookbook JunitChangingResult, da mesma forma que no exemplo anterior, a restrição que verifica se todo o Cookbook possui uma recipe chamada Main retornou *false*, para satisfazer a condição desta restrição foi definida a Recipe Main. Na linha 3 foi utilizado o comando de extensão de classes, a restrição aplicável a este comando é dinâmica, portanto esta verificação não foi realizada. Na linha 5 são criados novos atributos para a classe TestClass, neste caso a restrição OCL que verifica se estes atributos possuem nomes diferentes dos atributos já existentes é aplicada. Nas linhas 7 e 8 e de 12 a 17 são utilizados os comandos de extensão de métodos, como estes comandos possuem restrições dinâmicas, eles não foram avaliados, esta verificação fica sob responsabilidade do reutilizador do framework. Na linha 10 são definidos novos métodos para a classe TestClass, neste caso a restrição OCL que verifica se estes métodos possuem nomes diferentes dos métodos já existentes é aplicada corretamente.

6.4 Conclusões

Este capítulo apresentou a segunda contribuição deste trabalho: os testes da corretude das restrições OCL aplicáveis aos comandos RDL por meio da utilização da ferramenta USE. A ferramenta USE permite especificar sistemas de informação utilizando um subconjunto da linguagem UML e permite demonstrar as restrições aplicáveis sobre esses modelos através de expressões OCL que podem ser verificadas com o avaliador de expressões OCL que a ferramenta inclui.

O meta-modelo da linguagem RDL foi descrito e em seguida populado com os elementos correspondentes à infra-estrutura de componentes a serem avaliados. Foram utilizados 3 exemplos para testar a corretude dos resultados das expressões OCL. No primeiro caso foi utilizado um exemplo simples para avaliar todas as restrições estáticas referentes aos comandos RDL. Este objetivo foi alcançado, sendo que as restrições retornaram os resultados esperados corretamente, por exemplo, quando foi definida uma classe com o nome de uma classe já existente no modelo, a expressão OCL alertou a inconsistência desta definição. Os outros dois exemplos utilizados testaram a utilização das restrições estáticas e serviram para demonstrar que nos casos onde as restrições aplicáveis aos comandos são dinâmicas, ainda não é possível avaliar a corretude do modelo após a aplicação dos comandos RDL, sendo este um ponto a ser considerado no futuro.

7 CONCLUSÕES E TRABALHOS FUTUROS

A crescente exigência por sistemas confiáveis e de boa qualidade deu origem a muitas metodologias para o desenvolvimento de sistemas computacionais. Atualmente são utilizados métodos formais na especificação de sistemas com o objetivo de construir sistemas de forma mais sistemática; eles permitem descobrir e corrigir os problemas existentes por meio da modelagem do comportamento do sistema, quase sempre o comportamento funcional, procurando reduzir inconsistências e ambigüidades através das provas de propriedades.

Uma especificação deve apontar – de forma exata e não ambígua – todas as condições a que um objeto deve satisfazer. A especificação formal constitui-se numa coleção de diferentes técnicas, mas tendo em comum o uso da matemática para especificar o comportamento dos sistemas. Neste contexto, podemos fazer referência à linguagem UML, que possui um documento que constitui uma parte importante na definição padrão da linguagem [UML99]. O documento fornece uma descrição – em meta-modelo da linguagem – apresentada em termos de três visões: *(1)* sintaxe abstrata usando um subconjunto das notações de modelagem estática de UML; *(2)* regras de boa formação expressas na linguagem de restrições de objetos (OCL); e *(3)* semântica dos elementos de modelagem, descrita em linguagem natural. Neste sentido, a especificação da linguagem UML serviu de referência no desenvolvimento deste trabalho, cujo objetivo foi estender a formalização da linguagem de descrição de reuso (RDL).

As próximas seções apresentam um resumo do trabalho desenvolvido, as suas contribuições e indicações de trabalhos futuros.

7.1 Resumo do trabalho desenvolvido

O estudo da aplicação dos métodos formais salientou a importância do uso destes conceitos para aumentar a confiabilidade dos modelos de software. Com isso foi estudada a forma utilizada na especificação da linguagem UML para reduzir ambigüidades dos modelos criados pela linguagem. Também foram pesquisadas duas abordagens – Action Semantics e a linguagem BasicMTL – que utilizam os mesmos conceitos empregados pela UML na especificação formal: meta-modelo e linguagem de restrição de objetos. O uso da abordagem de meta-modelos e OCL na sintaxe abstrata da linguagem BasicMTL faz com que os modelos criados com a linguagem sejam mais corretos e confiáveis. Action Semantics propõe o uso de OCL para verificar se uma dada implementação respeita um conjunto de restrições em OCL;

esta proposta compartilha do mesmo objetivo da definição do meta-modelo RDL e do uso de restrições em OCL propostos nesta dissertação.

Após o estudo dos trabalhos relacionados, foi definida a forma a ser utilizada na extensão da formalização da linguagem RDL. Para viabilizar a inclusão de métodos formais na linguagem RDL, foi definido o uso da abordagem de meta-modelos. O processo de verificação ocorreu através das seguintes etapas:

- **Modelagem:** esta etapa consistiu em construir um modelo formal da linguagem a partir da BNF da RDL; foi derivado o meta-modelo RDL, compatível com o meta-modelo da linguagem UML; e foram definidos os possíveis comportamentos dos modelos criados a partir meta-modelo.
- **Especificação:** esta etapa consistiu em especificar as propriedades comportamentais desejáveis. Os comportamentos desejados dos modelos foram descritos por meio de um conjunto de restrições em linguagem natural, aplicáveis aos comandos disponibilizados pela linguagem RDL, que incorporam propriedades e comportamentos presentes na RDL. Estas restrições foram derivadas a partir de relatórios técnicos e manuais criados pelos autores da linguagem. Em seguida as restrições em linguagem natural foram mapeadas para a linguagem de restrição de objetos (OCL).
- **Verificação:** esta etapa consistiu em verificar se as especificações escritas foram satisfeitas pelos modelos criados. Para testar a utilização das restrições, o meta-modelo RDL foi descrito na ferramenta USE e foram executados alguns scripts RDL contendo os comandos para os quais foram definidas as restrições OCL. Foram criadas algumas situações incorretas propositalmente para avaliar a corretude das respostas das restrições.

7.2 Contribuições

Podemos concluir que as principais contribuições deste trabalho são:

- **Meta-modelo da linguagem RDL:** um diagrama de classes UML de acordo com a BNF definida anteriormente pelos autores da linguagem.
- **Conjunto de restrições em linguagem natural:** aplicáveis aos comandos disponibilizados pela linguagem, estas restrições incorporam propriedades e comportamentos presentes na RDL. As restrições em linguagem natural foram definidas para os comandos das seguintes tarefas: Tarefas Básicas de Programação, que envolvem comandos de criação de novas classes, métodos e atributos e definições de herança; Tarefas Específicas de Instanciação, que envolvem comandos de escolha de um elemento, estensão de classes, seleção de estensão de classes e estensão de métodos; Tarefas Específicas de Padrões, que

envolvem os comandos relacionados a processos de instanciação recorrentes; Tarefas Específicas de Seqüência, que envolvem a maneira que os comandos RDL são combinados no fluxo de execução e por fim as Tarefas Diversas, que incluem o comando de atribuição.

- **Conjunto de restrições OCL:** as restrições mapeadas para a linguagem OCL foram as restrições estáticas; exemplos de verificação estática são as verificações aplicáveis aos comandos de criação de classes, métodos, atributos e as restrições que verificam a existência de uma Recipe chamada Main nos modelos. As restrições dinâmicas foram descritas somente em linguagem natural, pois a responsabilidade pela corretude da implementação destes comandos deve ser verificada em tempo de execução e fica sob responsabilidade do reutilizador do framework.
- **Exemplos de utilização:** foram utilizados scripts RDL para testar a aplicabilidade das restrições definidas em OCL. Estes scripts foram executados na ferramenta USE, possibilitando a verificação da corretude do resultado das avaliações das restrições.

7.3 Trabalhos futuros

Pode-se destacar como uma atividade importante prevista para trabalhos futuros uma integração com o xFIT [OLI01] que existe hoje; isto poderá ser feito através da geração automática do script USE. Neste trabalho, uma dificuldade encontrada foi que os scripts RDL tiveram que ser descritos manualmente na ferramenta USE. Assim que houver uma integração, será mais fácil para o reutilizador do framework testar a corretude dos modelos RDL na ferramenta USE, sem que para isto seja necessário digitar comando por comando na ferramenta.

Outra atividade que poderia ser realizada é a inclusão das restrições OCL na linguagem RDL, assim, quando o reutilizador do framework quisesse avaliar os modelos criados, ele teria esta opção integrada à linguagem.

REFERÊNCIAS

[ALE99] Alencar, F. M. R. "**Mapeando a Modelagem Organizacional em Especificações Precisas**". PhD Thesis. Universidade Federal de Pernambuco, UFPE, 1999, 304 p.

[BOO99] Booch, G.; Jacobson, I.; Rumbaugh, J. "**Unified Modeling Language User Guide**". Rational Software Corporation. Addison-Wesley Object Technology Series, 1999.

[BOW94a] Bowen, J.; Hinchey, G. "**Seven More Myths of Formal Methods: Dispelling Industrial Prejudices**". In: M. Naftalin; T. Denvir and M. Bertran, FME´94: Industrial Benefit of Formal Methods. Springer-Verlag, LNCS 873, 1994, pp. 105-117.

[CLA96] Clarke, E. M.; Wing, J. M. "**Formal methods: State of the art and future directions**". Carnegie Mellon University (CMU), Tech. Rep. CMU-CS-96-178, Set. 1996. Capturado em: http://citeseer.ist.psu.edu/article/clarke96formal.html, Maio 2006.

[EVA99] Evans, A.; Lano, K., France, R., Rumpe, B. "**Meta-Modeling Semantics of UML".** In: Behavioural Specifications for Businesses and Systems. Kluwer Academic Publishers, Editor: Haim Kilov, 1999, cap. 4.

[FAY99] Fayad, M. E.; Schmidt, D. C; Johnson, R. "**Building application frameworks: Object-oriented foundations of framework design**". John Wiley & Sons. 1999.

[HER01] Herbsleb, J. D.; Moitra, D. "**Global Software Development** ". IEEE Software Magazine, IEEE Computer Society, EUA, Mar/Abr 2001.

[JOH97] Johnson, R. E. "**Components, frameworks, and patterns**". In: Proc. of the ACM Symposium on Software Reusability, (SST' 97), Boston, Maio 1997, pp. 17-20.

[JOH98] Johnson, R. E.; Foot B. "**Designing Reusable Classes**". Journal of Object Oriented Programming, 1 (2), 1998, pp. 22-35.

[JAC99] Jacobson, I.; Booch, G.; Rumbaugh, J. "**Unified Software Development Process**". Rational Software Corporation. Addison-Wesley Object Technology Series. Jan., 1999.

[KLE06] Kleppe, A. G.; Warmer, J. B. "**Unification of static and dynamic semantics of UML: A study in redefining the semantics of UML using the pUML OO meta modelling approach**". Tech. Rep. Klasse Objecten, 2003. Capturado em: http://www.klasse.nl, Janeiro 2006.

[MEY97] Meyer, B. "**Object-Oriented Software Construction**", 2nd edition, Prentice-Hall PTR, ISBN 0-13629155-4, 1997.

[MEN02] Menezes, P. B. "**Linguagens Formais e Autômatos***"*. Instituto de Informática UFRGS. 4ª Edição, 2002.

[MTL05] Basic MTL Manual. Capturado em: http://modelware.inria.fr/static_pages/docs/html.chunked/BasicMTL_UserManual/index.html, Julho 2006.

[OCL06] OCL 2.0 Specification. Capturado em: http://www.omg.org/docs/ptc/05-06-06.pdf, Fevereiro 2006.

[OLI01] Oliveira, T. C. "**Uma Abordagem Sistemática para a Instanciação de Frameworks Orientados a Objetos***"*. Ph.D. Thesis, Pontifícia Universidade Católica do Rio de Janeiro, 2001, 158 p.

[OLI04] Oliveira, T. C.; Alencar, P. S. C.; Lucena, C. J. P.; Cowan, D. D. "**Software Process Representation and Analysis for Framework Instantiation***"*. IEEE Trans. Softw. Eng. 30, 3, 2004, pp. 145-159.

[OLI05] Oliveira, T. C.; Alencar, P. S. C.; Cowan, D. D.; Mendonça, M. "**Assisting Framework Instantiation: Enhancements to Process-Language-based Approaches**". Technical Report CS-2005-25 at University of Waterloo, School of Computer Science. 2005.

[OLI07] Oliveira, T. C.; Alencar, P. S. C.; Lucena, C. J. P.; Cowan, D. D. "**RDL: A Language for Framework Instantiation Representation***"*. Journal of Systems and Software, In Press, Corrected Proof, Available online 20 January 2007.

[OMG99] OMG. UML Semantics. In: "OMG Unified Modeling Language Specification", Version 1.3, Jun. 1999.

[PLO97] Plösch, R. "**Design by Contract for Python**". IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference (APSEC97/ICSC97), Hongkong, 1997, pp. 2-5.

[PRE99] Pree, W. "**Hot-spot-driven development**". In Fayad, M.; Johnson, R.; Schmidt, D. "Building Application Frameworks: Object-Oriented Foundations of Framework Design", John Willey and Sons, 1999, pp. 379–393.

[RAM03] Ramalho, F., Robin, J.; Barros, R. S. M. "**XOCL . An XML language for specifying logical constraints in object oriented models**". In JUCS, Journal of the Universal Computer Science, 2003.

[RAS06] Rasmussen, R. W. "**A framework for the UML meta model**". Capturado em: http://www.ii.uib.no/~rolfwr/thesisdoc/main261.html, Janeiro 2006.

[RIC01] Richters, M. "**A Precise Approach to Validating UML Models and OCL Constraints***"*. Monographs of the Bremen Institute of Safe Systems, 2001.

[RUM99] Rumbaugh, J.; Jacobson, I.; Booch, G. "**Unified Modeling Language Reference Guide**". Rational Software Corporation. Addison-Wesley Object Technology Series, 1999.

[SOM05] Sommerville, I. "**Engenharia de Software**". 6ª Edição. Addison-Wesley. USA, 2005, pp. 400-401.

[SUN01] Sunyé, G.; Pennaneac'h, F.; Ho, W-M.; Guennec, A. Le; Jézéquel, J-M. "**Using UML Action Semantics forExecutable Modeling and Beyond. Conference on Advanced Information Systems Engineering**". 2001.

[TRA01] Travassos, G. H.; Shull, F.; Carver, J. "**Working with UML: A software design process based on inspections for the unified modeling language".** Advances in Computers, San Diego, v. 54, n. 1, 2001, pp. 35-97.

[UML99] OMG. "**UML Semantics**". In OMG Unified Modeling Language Specification, Version 1.3, 1999, Cap. 2.

[USE06] USE. Capturado em: http://dustbin.informatik.uni-bremen.de/projects/USE, Julho 2006.

[VAC01] Vaccare Braga, R. T.; Masiero, P. C. "**A Pattern Language-based Approach for Framework Construction and Instantiation**". Position Paper, 1st ASERC Workshop on Software Architecture, Computing Science Centre University of Alberta, Edmonton, Alberta, Canadá, 2001, pp. 24-25.

[WAR99] Warmer, J. B.; Kleppe, A. G. "**The Object Constraint Language: Precise Modeling with UML**". Addison-Wesley. USA, 1999.

[WIR98] Wiryana, M. "**Information system development: an interdiscipline approach**". Capturado em: http://wiryana.pandu.org/artikel/paper_issm/, Outubro 2006.

ANEXO I – BNF DA LINGUAGEM RDL (SEÇÃO 1.1)

COOKBOOK ::=	**Cookbook** NAME IP_RECIPE* IP_MAIN **end_cookbook;**
IP_RECIPE ::=	[COMMENT_EXP] **recipe** IP_NAME; IP_RECIPE_BODY **end_recipe**;
IP_RECIPE_BODY ::=	IP_CMD*
IP_MAIN ::=	**recipe main** IP_RECIPE_BODY **end_recipe;**
IP_NAME ::=	NAME
IP_CMD ::=	IP_ASSIGN_CMD ;|IP_EXP_CMD; |IP_LOOP_CMD;
IP_ASSIGN_CMD::=	NAME = IP_BASIC
IP_LOOP_CMD ::=	**loop** IP_RECIPE_BODY **end_loop**
IP_EXP_CMD ::=	IP_TASK | IP_TASK # IP_TASK | IP_TASK o IP_TASK | IP_TASK || IP_TASK
IP_TASK ::=	[COMMENT_EXP] IP |[COMMENT_EXP] IP_NAME
IP ::=	IP_BASIC REQUIRES_EXP*
IP_BASIC ::=	IP_CLASS | IP_METHOD | IP_ATTRIBUTE | IP_ELEMENT
IP_CLASS ::=	**new_class |** **class_extension** (CLASS_EXP) | **selection_class_extension** (CLASS_EXP) | **pattern_class_extension** (CLASS_EXP , NAME,LIST) | **interface_implementation** (CLASS_EXP , INTERFACE_EXP)
IP_METHOD ::=	**new_method** (CLASS_EXP) | **method_extension** (CLASS_EXP, CLASS_EXP, METHOD_EXP) | **pattern_method_extension** (CLASS_EXP, CLASS_EXP, METHOD_EXP , NAME, LIST)
IP_ATTRIBUTE ::=	**new_attribute (**CLASS_EXP) **value_selection** (CLASS_EXP, ATTRIB_EXP, LIST) | **value_assignment** (CLASS_EXP, ATTRIB_EXP)
IP_ELEMENT::=	**element_choice** (ELEMENT)
ELEMENT::=	CLASS_EXP | METHOD_EXP | ATTRIB_EXP
CLASS_EXP ::=	NAME
METHOD_EXP ::=	NAME
ATTRIB_EXP ::=	NAME
INTERFAC_EXP ::=	NAME
LIST::=	(LIST_EXP)
LIST_EXP::=	STRING_EXP , LIST_EXP | STRING_EXP
NAME::=	IDENTIFIER_EXP
REQUIRES_EXP::=	**requires** ORDER_EXP | **requires** ELEMENT
ORDER_EXP::=	**before** IP_TASK |**after** IP_TASK | **sync** IP_TASK | **exclusive** IP_TASK
COMMENT_EXP ::=	// STRING_EXP | COMMENT_EXP
STRING_EXP ::=	**String**

ANEXO II – DESCRIÇÃO TEXTUAL DO META-MODELO RDL (SEÇÃO 6.1.2)

```
model MetaRDL
-- Classes do Meta modelo RDL

class ReusableAsset
attributes
      name : String
end

class InputModel
attributes
      name : String
end

class OutputModel
attributes
      name : String
operations
      new_class(Cname: String): Class
      class_ext(c: String, newclass: String): Class
end

class Cookbook
attributes
      name : String

end

class Recipe
attributes
      name : String
end

class Pattern
attributes
      name : String
end

class Body
attributes
      comm : Command
end

class Command
attributes
      name : String
end

class Parameter
attributes
      name : String
end

class Class
attributes
      name : String
      meth : Method
      att  : Attribute
operations
      new_att(c: Class, attName: Attribute): Class
      new_meth(c: Class, metName: Method): Class
      new_inh(superC: Class, subC: Class): Class
end

class Attribute
attributes
      name : String
end

class Method
attributes
      name : String
end

-- Associations
association RC between
```

```
      ReusableAsset[1..*]
      Cookbook[1..*]
end

association RI between
      ReusableAsset[1..1]
      InputModel[1..1]
end

association IC between
      InputModel[1]
      Class[1..*]
end

association RO between
      ReusableAsset[1..1]
      OutputModel[1..1]
end

association OC between
      OutputModel[1]
      Class[1..*]
end

association CR between
      Cookbook[1..*]
      Recipe [1..*]
end

association PR between
      Pattern [*]
      Recipe [*]
end

association RP between
      Recipe [1..*]
      Parameter [*]
end

association RB between
      Recipe [0..1]
      Body[0..1]
end

association BC between
      Body [1..*]
      Command [1..*]
end

association CP between
      Command [0..*]
      Parameter[0..*]
end

association PC between
      Parameter[*]
      Class[*]
end



association SuperSub between
      Class[0..*] role super
      Class[0..*] role sub
end

association CA between
      Class[0..*]
      Attribute[0..*]
end

association CM between
      Class[0..*]
      Method[0..*]
end

-- Constraints (Restrições aplicáveis ao meta modelo)

constraints
context Cookbook inv CookbookEmpty: self.name <> ''
--context Cookbook inv selfRecName: self.rec.name <> ''

context Cookbook inv CookbookHasMain:
self.recipe->exists(r|r.name='Main')

context Recipe inv RecipeNameUnique: Recipe.allInstances-> forAll
(other|self.name = other.name implies self=other)
```

```
context Parameter inv ParameterNameUnique: Parameter.allInstances->
forAll (other|self.name = other.name implies self=other)

context Class inv ClassNameUnique: Class.allInstances-> forAll
(other|self.name = other.name implies self=other)

context Method inv MethodNameUnique: Method.allInstances-> forAll
(other|self.name = other.name implies self=other)

context Class inv AttributeNameUnique: Attribute.allInstances->
forAll (other|self.name = other.name implies self=other)

context Cookbook inv AttributesDefined: name.isDefined

context Recipe inv AttributesDefined: name.isDefined

context Class inv AttributesDefined: name.isDefined

context Method inv AttributesDefined: name.isDefined

context Attribute inv AttributesDefined: name.isDefined

context OutputModel:: new_class(Cname: String): Class
      pre: OutputModel.allInstances-> forAll
            (other|self.name = other.name implies self=other) and self.name <> Cname

--self.recipe->exists(r|r.name='Main')
context Class:: new_att(c: Class, attName: Attribute): Class
      pre: (self.name <> c.name) and (self.name <> attName.name)

context Class:: new_meth(c: Class, metName: Method): Class
      pre: (self.name <> c.name) and (self.name <> metName.name)

context Class:: new_inh(superC: Class, subC: Class): Class
      pre: (self.name <> superC.name) and (self.name <> subC.name) and
           (subC.name < superC.name)
```

ANEXO III – SCRIPT DA FERRAMENTA USE (SEÇÃO 6.1.3)

```
1   !create Y : Recipe
2   !create X : Cookbook
3   !set X.name := 'Cook1'
4   !set Y.name := 'Main'
5   !insert (X,Y) into CR
6   !create A : Recipe
7   !set A.name := 'Main'
8   !create B : Body
9   !create IM : InputModel
10  !set IM.name := 'Modelo de Entrada'
11  !create RA : ReusableAsset
12  !set RA.name := 'ArtefatoReutilizavel'
13  !insert (RA,IM) into RI
14  !insert (RA,X) into RC
15  !create C : Command
16  !set C.name := 'New_Class'
17  !insert (B,C) into BC
18  !insert (Y,B) into RB
19  !create P : Parameter
20  !set P.name := 'P1'
21  !insert (C,P) into CP
22  !create Carro : Classe
23  !set Carro.name := 'Pálio'
24  !insert (P,Carro) into PC
25  !create Emp : Classe
26  !set Emp.name := 'Empregado'
27  !create Pessoa : Classe
28  !set Pessoa.name := 'Pessoa'
29  !create OM : OutputModel
30  !set OM.name := 'Output Model'
31  !insert (RA,OM) into RO
32  !insert (P,Emp) into PC
33  !insert (P,Pessoa) into PC
34  !insert (OM,Carro) into OC
35  !openter OM new_class(Aviao)
36  !openter OM class_ext(Carro,Aviao)
37  !create a : Attribute
38  !set a.name := 'C1'
39  !create b : Attribute
40  !set Emp.att:= a
41  !set b.name := 'C1'
42  !openter Carro new_att(Aviao,a)
43  !create t : Method
44  !set t.name := 'acelera'
45  !openter Carro new_meth(Carro ,t)
46  !openter Carro new_inh(Carro ,Pessoa)
```

ANEXO IV – SCRIPT DA FERRAMENTA USE (SEÇÃO 6.1.4)

```
1.  !create X : Cookbook
2.  !create Y : Recipe
3.  !set X.name := 'JunitFixture'
4.  !set Y.name := 'Main'
5.  !insert (X,Y) into CR
6.  !create B : Body
7.  !create IM : InputModel
8.  !set IM.name := 'Modelo de Entrada'
9.  !create OM : OutputModel
10. !set OM.name := 'Output Model'
11. !create RA : ReusableAsset
12. !set RA.name := 'ArtefatoReutilizavel'
13. !insert (RA,IM) into RI
14. !insert (RA,OM) into RO
15. !insert (RA,X) into RC
16. !create Co : Command
17. !set Co.name := 'New_Class'
18. !insert (B,Co) into BC
19. !insert (Y,B) into RB
20. !create P : Parameter
21. !set P.name := 'P1'
22. !insert (Co,P) into CP
23. !create C : Classe
24. !set C.name := 'testClass'
25. !openter OM class_ext(C,?)
26. !openter C new_att(C,?)
27. !openter C new_meth(C,?)
```

ANEXO V – SCRIPT DA FERRAMENTA USE (SEÇÃO 6.1.5)

```
1.  !create X : Cookbook
2.  !create Y : Recipe
3.  !set X.name := 'JunitChangingResult'
4.  !set Y.name := 'Main'
5.  !insert (X,Y) into CR
6.  !create B : Body
7.  !create IM : InputModel
8.  !set IM.name := 'Modelo de Entrada'
9.  !create OM : OutputModel
10. !set OM.name := 'Output Model'
11. !create RA : ReusableAsset
12. !set RA.name := 'ArtefatoReutilizavel'
13. !insert (RA,IM) into RI
14. !insert (RA,OM) into RO
15. !insert (RA,X) into RC
16. !create C : Command
17. !set C.name := 'New_Class'
18. !insert (B,C) into BC
19. !insert (Y,B) into RB
20. !create P : Parameter
21. !set P.name := 'P1'
22. !insert (C,P) into CP
23. !create C : Classe
24. !set C.name := 'testClass'
25. !openter C class_ext(C,?)
26. !openter C new_att(C,?)
27. !openter C new_meth(C,?)
28. !create C2 : Classe
29. !set C2.name := 'resultClass'
30. !openter C2 class_ext(C2,?)
```