Validação de modelos OntoUML enriquecidos com restrições OCL (Portuguese-BR)

Thesis · May 2013
DOI: 10.13140/RG.2.1.4414.6729

1 author:

John Guerson
Universidade Federal do Espírito Santo
7 PUBLICATIONS **29** CITATIONS

SEE PROFILE

Universidade Federal do Espírito Santo

John Guerson

Validação de modelos OntoUML enriquecidos com restrições OCL

Vitória - ES, Brasil
2013

John Guerson

Validação de modelos OntoUML enriquecidos com restrições OCL

Monografia apresentada ao curso de Ciência da Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. João Paulo Andrade Almeida

Vitória - ES, Brasil

2013

AGRADECIMENTOS

Agradeço a Deus pelo sustento nessa difícil jornada e por esta tão sonhada conquista a qual significa muito pra mim.

Sou grato à minha querida Ghiselle - In Memoriam - pelo seu amor e enorme carinho e dedicação aos quais foram essenciais pra mim.

Aos meus pais, Nélio e Cristina, e as minhas irmãs mais novas Susie, Dianne e a pequenina Cindy, que participam comigo dessa realização, fruto de muita luta e dedicação.

Agradeço também aos meus familiares por acreditarem em mim e desempenharem um importante papel pra que eu pudesse chegar até aqui: aos meus tios Carlos e Dayse, aos meus padrinhos Wagner e Aparecida, as minhas tias Odete e Maria, minha avó Socorro, meu tio Arnaldo, meus primos Leandro e Leonardo, e finalmente meus tios Zé – In Memoriam – e Maria.

Sou grato também ao meu orientador João Paulo Almeida e sua esposa Patrícia Dockhorn pela oportunidade, por todo o aprendizado e compreensão nos momentos adversos.

Aos meus estimados colegas do laboratório NEMO por toda a convivência e aprendizado compartilhados durante todo esse período. Em especial, Tiago Prince, Julio Nardi, Antognoni Albuquerque e Ernani Santos, os quais contribuíram significativamente para o amadurecimento do trabalho, quer seja diretamente ou indiretamente através de discussões, dúvidas sanadas e conversas.

RESUMO

Um modelo conceitual é uma forma de representar o que os modeladores ou *stakeholders* percebem de uma porção da realidade do mundo físico e social com o intuito de apoiar o entendimento (aprendizado), solução de problemas e comunicação sobre um domínio. Para uma representação de modelos conceituais, é necessária uma linguagem de modelagem conceitual capaz de apoiar a definição de modelos que capturem essa porção da realidade, i.e., conceituação de acordo com a intenção do modelador, ou, conceituação pretendida. A linguagem de modelagem OntoUML foi proposta visando capturar essa conceituação pretendida permitindo distinções detalhadas entre diferentes tipos de classes de acordo com a ontologia de fundamentação UFO.

A tarefa de modelagem é desafiadora uma vez que é necessário julgar a qualidade dos modelos produzidos avaliando se eles refletem ou não a conceituação de acordo com a intenção do modelador. Uma abordagem pra facilitar o processo de avaliação (também chamado de validação) é a simulação visual das instâncias do modelo OntoUML através de Alloy confrontando o modelador ou *stakeholder* com as possibilidades de instanciação do modelo.

Essa abordagem de validação revela que a linguagem de modelagem OntoUML não é expressiva o suficiente para representar todos os aspectos relevantes da conceituação pretendida, uma vez que não são diretamente representadas em OntoUML muitas restrições de domínio.

Esta monografia propõe uma abordagem para a inclusão de restrições de domínio utilizando a linguagem OCL (*Object Constraint Language*) na validação de modelos OntoUML e na simulação visual de suas instâncias através de Alloy. Neste trabalho, apresentamos a abordagem de validação baseada em restrições OCL desenvolvida através de uma ontologia em OntoUML de parte de um domínio de acidente de trânsito.

ABSTRACT

A conceptual model is a means to represent what modelers perceive in a portion of reality of the physical and social world with the purpose of supporting the understanding (learning), problems solving and communication about a domain. In order to represent such conceptual models, a conceptual modeling language must be able to capture the modeler's intended conceptualization. The OntoUML modeling language was proposed with this aim, enabling modelers to express finer-grained distinctions between different types of classes according to the foundation ontology UFO.

The modeling task is a challenging activity since it is necessary to judge the quality of the produced models assessing whether or not they reflect the intended conceptualization. An approach to facilitate model assessment (in particular validation) is the visual simulation of OntoUML model instances through Alloy confronting the modeler, or stakeholder, with the instantiation possibilities of the model.

The validation approach reveals that the OntoUML modeling language is not expressive enough to represent all the relevant aspects of the intended conceptualization. The instances generated by the visual simulation reflect the necessity of capturing additional domain constraints in order to complement the OntoUML model.

This monograph proposes an approach to include domain constraints, using the OCL Language (Object Constraint Language), into the validation of OntoUML models and the visual simulation of its instances through Alloy. We illustrate the validation approach using a road traffic accident ontology in OntoUML enriched with OCL constraints.

Lista de Figuras

Lista de Tabelas

Conteúdo

1 Introdução

Este capítulo apresenta a motivação deste projeto (1.1), apresenta seus objetivos (1.2), descreve a abordagem utilizada (1.3) e por fim apresenta a estrutura deste relatório (1.4).

1.1 Motivação

John Mylopoulos (1992) define a disciplina de modelagem conceitual como "a atividade de descrever formalmente alguns aspectos do mundo físico e social em torno de nós com propósitos de entendimento e comunicação. Modelagem conceitual suporta recursos de estruturação e inferência que são fundamentados psicologicamente. Afinal de contas, as descrições que surgem de atividades de modelagem conceitual têm a intenção de serem usadas por humanos, e não por máquinas...".

Um modelo conceitual, de acordo com essa visão, é uma forma de representar o que os modeladores (ou *stakeholders* representados aqui também por modeladores) percebem de uma porção da realidade do mundo físico e social, usado para apoiar o entendimento (aprendizado), solução de problemas, e comunicação entre especialistas de um domínio, em outras palavras, uma forma de expressar suas conceituações (GUIZZARDI, 2005) sobre um universo de discurso (BENEVIDES et al., 2011). Assim, uma vez que um nível suficiente de entendimento e concordância sobre o domínio foi atingido, o modelo conceitual pode ser usado nas fases subsequentes do processo de desenvolvimento de software (GUIZZARDI, 2005).

Uma linguagem de modelagem conceitual deve apoiar a definição de modelos que capturem a conceituação de domínio de acordo com a intenção do modelador. Esta preocupação justificou a revisão de uma porção da UML (*Unified Modeling Language*) (OMG, 2005, UML 2.0) em uma linguagem de modelagem conceitual chamada OntoUML (GUIZZARDI, 2005). Essa revisão permite ao modelador fazer distinções detalhadas entre diferentes tipos de classes de acordo com a ontologia de fundamentação UFO (GUIZZARDI, 2005).

A Figura 1 ilustra a conceituação pretendida, a definição do modelo conceitual OntoUML de acordo com essa conceituação, e a interpretação desse modelo por stakeholders. Os círculos de cor branca representam o modelador e os stakeholders. O retângulo de cor branca representa o modelo conceitual OntoUML e a nuvem de cor cinza a conceituação de acordo com a intenção do modelador.

Assim, a Figura 1 especifica a definição de um modelo conceitual OntoUML de acordo com a concepção do modelador e a capacidade de comunicação e entendimento desse modelo por stakeholders, ou seja, os stakeholders devem ser capazes de interpretar o modelo conceitual de acordo com a conceituação de domínio.

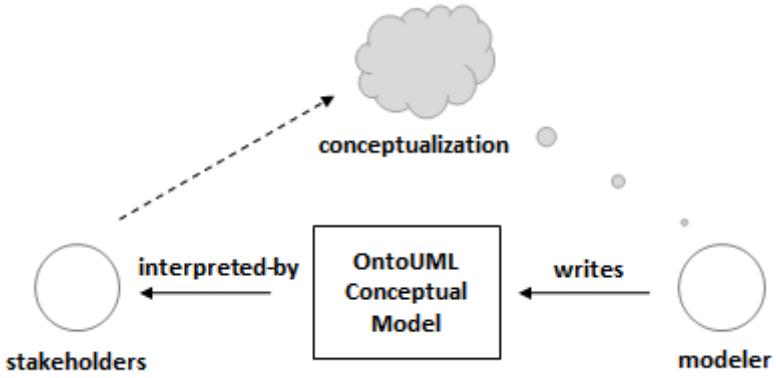

Figura 1. Conceituação e modelo conceitual OntoUML.

Apesar dos avanços na linguagem de modelagem, a tarefa de modelagem conceitual permanece desafiadora. Em uma abordagem madura de modelagem conceitual, é necessário que os modeladores tenham mecanismos para julgar a qualidade dos modelos conceituais produzidos, avaliando se esses modelos refletem, com a maior *acurácia* possível, a conceituação do domínio (BENEVIDES et al., 2011). A Figura 2 ilustra essa avaliação de modelos de acordo com a acurácia à conceituação de domínio. O círculo de cor azul claro representa os estados admissíveis de acordo com o modelo conceitual enquanto que a nuvem de cor cinza claro a conceituação de acordo com a concepção do modelador, i.e., os estados admissíveis de acordo com a conceituação de domínio.

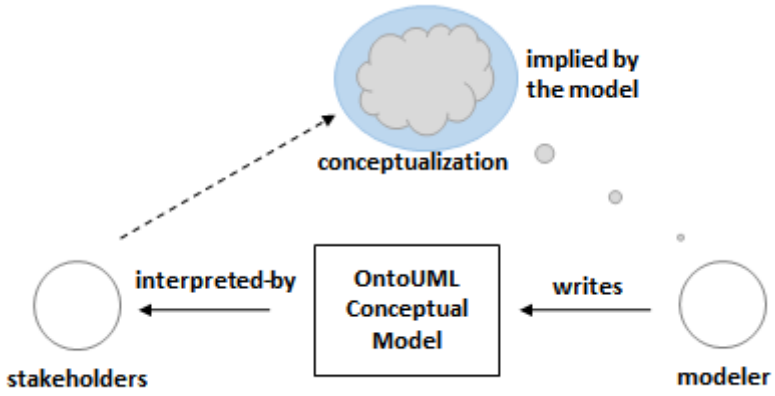

Figura 2. Avaliação da acurácia de modelos OntoUML.

A *acurácia* de modelos conceituais baseados em ontologia, incluindo-se aqueles expressos em OntoUML, pode ser avaliada com relação aos seus níveis de precisão (*precision*) e cobertura (*coverage*) com relação à conceituação de domínio (GUARINO, 2004).

A Figura 3**,** adaptada de (GUARINO, 2004), ilustra os conceitos de precisão e cobertura. O círculo branco pontilhado representa os estados admissíveis dos objetos da realidade. O círculo de cor cinza escuro representa os estados admissíveis de acordo com a conceituação do domínio, i.e., a porção da realidade de acordo com a intenção do modelador e o círculo de cor azul claro representa os estados admissíveis de acordo com o modelo conceitual.

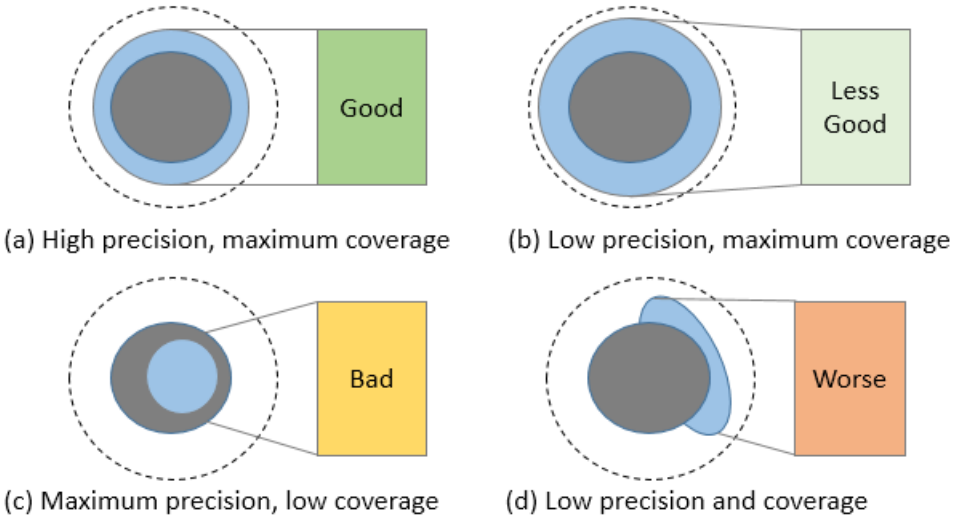

Figura 3. Ilustrando acurácia: precisão e cobertura.

Assim, a Figura 3 especifica a comparação entre a ontologia especificada pelo modelo (círculos de cor azul claro) e a conceituação de acordo com a intenção do modelador (círculos de cor cinza escuro).

A figura 3 (a) ilustra o que seria um modelo com uma boa acurácia: uma precisão alta e uma cobertura máxima com relação à conceituação de domínio.

A figura 3 (b) ilustra um modelo menos acurado: cobertura máxima porém com uma precisão baixa, tipicamente um modelo pouco restrito (*underconstrained*). Este modelo admitiria estados de objetos que são inadmissíveis na concepção do modelador.

Já a figura 3 (c) ilustra um modelo que possui uma precisão máxima porém uma cobertura baixa, tipicamente um modelo super-restrito, além do pretendido (*overconstrained*). Este modelo não permitiria estados de objetos que são considerados admissíveis na concepção do modelador.

Por último, a figura 3 (d) ilustra um modelo que possui uma precisão baixa e também uma cobertura baixa, tipicamente incluindo os problemas dos dois casos anteriores, considerado o pior dos casos e assim o menos desejado.

No contexto de processo de desenvolvimento de software, modelos não acurados, como os modelos das figuras (b), (c) e (d), levam a produção de implementações indesejadas. No contexto da modelagem conceitual em si, esses modelos prejudicam o entendimento, a concordância e o aprendizado sobre o domínio.

A abordagem proposta por Benevides (2011) e Braga (2010) para apoiar o processo de validação de modelos conceituais em OntoUML, ou seja, avaliar se os modelos são uma representação acurada da conceituação de domínio, é transformar esses modelos conceituais OntoUML em especificações na linguagem Alloy (JACKSON, 2012) e usar o analisador Alloy para gerar instâncias do modelo de forma visual confrontando o modelador com as possibilidades de instanciação. A Figura 4 mostra essa abordagem de validação via Alloy.

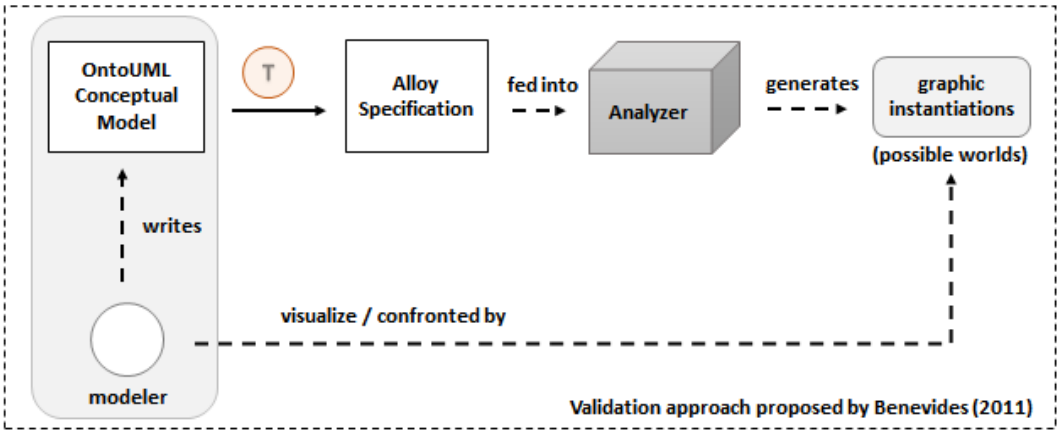

Figura 4. Abordagem de validação de modelos OntoUML via Alloy.

Atualmente existem duas transformação OntoUML para Alloy, a saber a de Benevides (2011) e a de Braga (2010). A abordagem de Braga (2010) define uma transformação que especifica uma estrutura de tempo linear e prioriza o desempenho na análise. A abordagem de transformações de OntoUML para Alloy referida neste trabalho é a de Benevides (2011) cuja transformação especifica uma estrutura temporal de mundos em Alloy viabilizando a formalização dos aspectos modais de OntoUML.

Alloy foi criada por Daniel Jackson, pelo grupo de projeto de software do MIT (*Massachusetts Institute of Technology*). É uma linguagem declarativa com uma lógica simples e expressiva, baseada na noção de relações e na lógica de primeira ordem para descrever e explorar estruturas. O seguinte trecho extraído de (JACKSON, 2012 p. XIII) nos remete bem ao que a linguagem Alloy proporciona:

> A experiência de se explorar o modelo com um analisador automático é emocionante e humilhante. A maioria dos modeladores já teve o benefício de ter os seus modelos revisados por colegas; com certeza é um bom meio de se encontrar falhas. Entretanto, poucos modeladores tiveram a experiência de submeter seus modelos a uma revisão automática e contínua. Construir um modelo de forma incremental com um analisador, executando simulações e checando propriedades à medida que se vai progredindo na modelagem, é uma experiência muito diferente do que usar apenas papel e caneta. A primeira reação tende a ser espanto: modelagem é muito mais divertida quando se tem um instantâneo re-

> torno visual. Quando você simula um modelo parcial, você vê exemplos que sugerem que novas restrições devem ser adicionadas.
>
> O sentimento de humilhação vem a medida que você descobre que quase nada você pode fazer certo. O que você escreve não significa exatamente aquilo que você pensava que significaria. E quando significa, não possui as consequências que você esperava. Ferramentas de análise automática são muito mais implacáveis do que revisores humanos. Eu agora me assusto com o pensamento de todos os modelos que eu escrevi (e publiquei) e que nunca foram analisados. Lenta, mas seguramente, a ferramenta te ensina a cometer menos e menos erros. Seu senso de confiança em sua habilidade de modelagem (e nos seus modelos) cresce!

Uma das limitações contidas na abordagem proposta por Benevides (2011) e Braga (2010) de validação de modelos OntoUML via Alloy é que a linguagem de modelagem conceitual OntoUML não é expressiva o suficiente para representar todos os aspectos relevantes da conceituação de domínio de acordo com a intenção do modelador. Isso é verificado através das possíveis instâncias geradas pelo analisador Alloy que não só confrontam o modelador quanto às decisões de modelagem, mas revelam a falta de restrições de domínio através de possibilidades de instanciação que são inadmissíveis na concepção do modelador.

Portanto, a linguagem OntoUML necessita de uma linguagem para a representação de restrições de domínio a fim de complementar sua notação diagramática. Ou seja, existe a necessidade de se descrever restrições adicionais no modelo conceitual OntoUML de forma a torná-lo mais acurado no que se refere à conceituação de domínio e de validar o modelo OntoUML adicionado de restrições via Alloy.

1.2 Objetivo

Este projeto tem como objetivo desenvolver uma abordagem para a inclusão de restrições de domínio na abordagem de validação de modelos OntoUML via Alloy, viabilizando o uso de restrições de domínio na abordagem de validação via Alloy existente e

fornecendo uma estratégia de validação baseada em restrições, complementando assim o trabalho de Benevides (2011).

1.3 Abordagem

Para especificar e validar as restrições de domínio em modelos conceituais OntoUML e assim representar modelos mais precisos, foi escolhida a linguagem de restrições de objetos OCL (*Object Constraint Language*) (OMG, 2012, OCL 2.3).

OCL é um padrão do grupo OMG (*Open Management Group*) para criar modelos, criar linguagens e transformar modelos. É uma linguagem *tipada*, declarativa e sem efeitos colaterais, baseada na teoria dos conjuntos e na lógica de predicados.

A linguagem é bastante semelhante à Alloy no sentido de que ambas são baseadas em lógica de primeira ordem, suas expressões não possuem efeitos colaterais e ainda, ambas são linguagens declarativas, ou seja, expressam *o que* a restrição representa (declarativa) e não *como* essa restrição é implementada (operacional).

OCL também faz parte do padrão UML sendo assim amplamente utilizada por organizações e comunidades científicas. A linguagem OCL permite a definição de expressões que capturam restrições, i.e., *constraints,* em modelos UML complementando a notação diagramática.

A Figura 5 ilustra a abordagem proposta neste trabalho, i.e., o desenvolvimento de uma transformação de um subconjunto de OCL para Alloy compatível com a transformação existente de OntoUML para Alloy e a sua inclusão na abordagem de validação de modelos OntoUML via Alloy proposta por Benevides (2011).

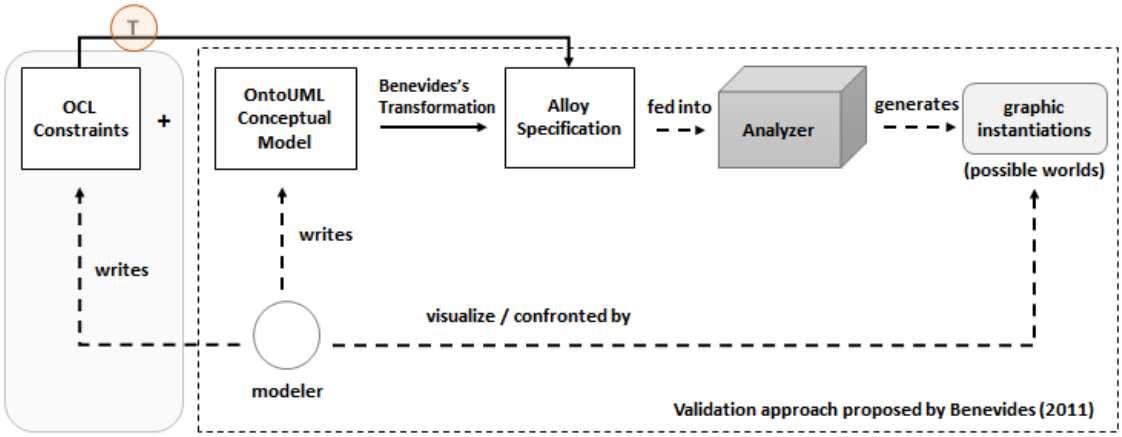

Figura 5. Inclusão de restrições OCL na abordagem de validação via Alloy.

Este projeto foi inspirado pelo trabalho de Anastasakis (2008) no qual a linguagem OCL foi utilizada como uma linguagem de restrições em modelos UML para fins de validação através da linguagem Alloy e de seu analisador (*UML2Alloy*, 2009). Em (ANASTASAKIS et al., 2008) foi definida uma transformação de um subconjunto de OCL para Alloy, incorporada na transformação de UML para Alloy, viabilizando a validação de modelos UML enriquecidos com restrições OCL via Alloy.

O subconjunto de OCL considerado neste trabalho é determinado pela expressividade e significância à OntoUML. Com isso, complementamos o trabalho de (BENEVIDES et al., 2011) viabilizando o uso de restrições em modelos conceituais OntoUML.

1.4 Estrutura do Trabalho

O restante deste trabalho está estruturado da seguinte forma:

Capítulo 2 – Referencial Teórico: Apresenta brevemente a linguagem OntoUML e descreve um exemplo OntoUML usado no decorrer deste trabalho; apresenta Alloy e por fim, a abordagem de validação de modelos OntoUML por meio da transformação do exemplo OntoUML para Alloy.

Capítulo 3 – A Linguagem de Restrições de Objetos OCL: Descreve a linguagem de restrições OCL e o subconjunto de restrições considerados para validação por meio da definição de restrições no exemplo OntoUML.

Capítulo 4 – A Transformação de OCL para Alloy: Define a transformação das restrições OCL e seus mapeamentos para Alloy compatível com a transformação de OntoUML para Alloy.

Capítulo 5 – Validação Enriquecida com Restrições OCL: Descreve a abordagem proposta para a validação de modelos OntoUML com restrições de domínio OCL.

Capítulo 6 – Implementação: Apresenta a implementação da transformação de OCL para Alloy desenvolvida e a incorporação da implementação na ferramenta experimental de validação de modelos MOVE.

Capítulo 7 – Conclusões: Apresenta as conclusões do trabalho, contribuições, limitações, trabalhos relacionados e trabalhos futuros.

2 Referencial Teórico

Este capítulo descreve o referencial teórico para este projeto, envolvendo OntoUML, Alloy e a transformação e validação de modelos OntoUML em Alloy.

A seção 2.1 introduz a linguagem OntoUML através de um exemplo de parte de um domínio de Acidente de Trânsito. A seção 2.2 apresenta Alloy. A seção 2.3 descreve a transformação de OntoUML para Alloy e apresenta a validação via Alloy baseada no trabalho de Benevides (2011) (sem restrições de domínio).

2.1 A Linguagem Ontologicamente Bem Fundamentada OntoUML

OntoUML (GUIZZARDI, 2005) é uma linguagem de modelagem para a criação de modelos conceituais ontologicamente bem fundamentados. A linguagem compreende diversas distinções ontológicas e incorpora regras sintáticas que refletem axiomas da Ontologia de Fundamentação UFO (*Unified Foundational Ontology*). O metamodelo da linguagem é isomórfico a uma parte da UFO, i.e., OntoUML contém como primitivas de modelagem um conjunto de tipos da ontologia de fundamentação.

A linguagem OntoUML foi proposta como uma extensão *lightweight* de UML. Possuindo um metamodelo mais rico que a UML, o modelador é capaz de fazer distinções mais refinadas entre os diferentes tipos de classes segundo a ontologia de fundamentação UFO (GUIZZARDI, 2005, p.107), produzindo modelos mais acurados com respeito à conceituação de domínio (GUIZZARDI, 2007). OntoUML tem sido aplicada com sucesso em projetos industriais de diversas áreas como: Petróleo e Gás (GUIZZARDI et al., 2010), Telecomunicações (BARCELOS, 2011), Eletrofisiologia do Coração (ECG) (GONÇALVES, 2010), e etc.

2.1.1 Exemplo em OntoUML

A Figura 6 apresenta um pequeno exemplo em OntoUML que representa parte de uma ontologia de domínio sobre acidentes de trânsito (*Road Traffic Accident*). Um acidente

de trânsito envolve veículos acidentados (*Crashed Vehicle*), vítimas (*Victim*) e ocorre em uma estrada (*Roadway With Accident*). Um acidente, neste nosso exemplo, pode ser do tipo colisão traseira (*Rear End Collision*), ao qual envolve exatamente dois veículos, um acidente em que um veículo bate na traseira de outro veículo. Além disso, neste domínio, uma vítima é sempre um viajante (*Traveler*), podendo ser passageiro (*Passenger*) ou motorista (*Driver*), que participa/faz uma viagem (*Travel*) em um determinado veículo (*Traveling Vehicle*). Não obstante, pessoas podem estar vivas (*Living Person*) ou mortas (*Deceased Person*).

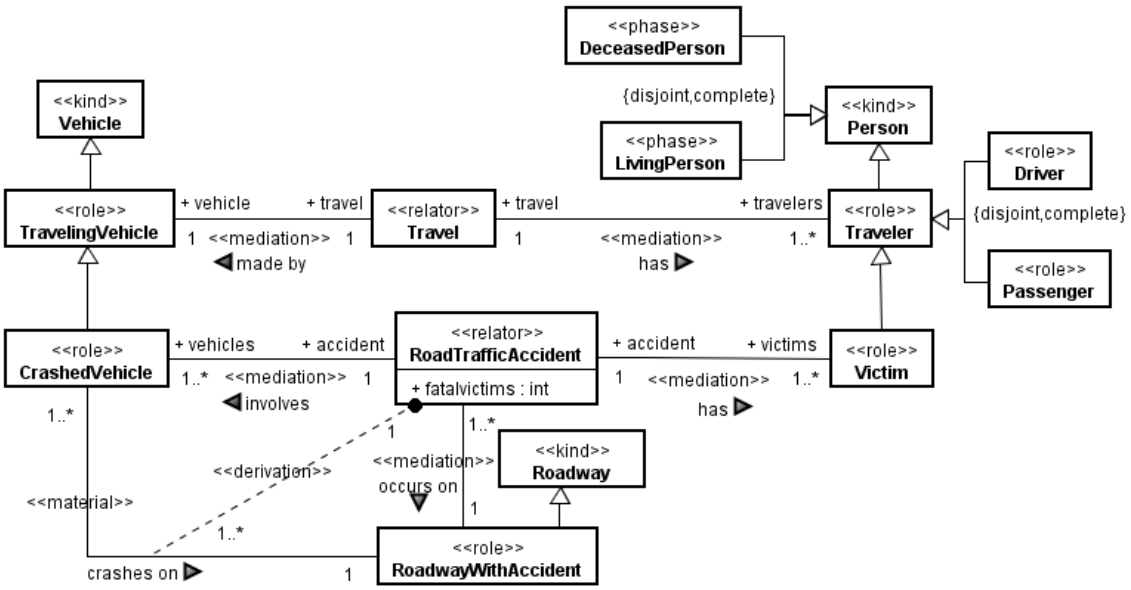

Figura 6. Modelo conceitual OntoUML de um domínio de acidente de trânsito.

Algumas das distinções ontológicas propostas pela ontologia de fundamentação são representadas como estereótipos na linguagem. Por exemplo, uma classe estereotipada como *<<kind>>* representa um conceito rígido (i.e., uma Pessoa não pode deixar de ser uma Pessoa sem deixar de existir) e provê um princípio de identidade para suas instâncias; um *<<role>>* por sua vez é um conceito antirrígido (i.e., um Passageiro pode deixar de ser Passageiro e ainda sim existir) e define o papel que uma instância de *<<kind>>* desempenha em um relacionamento (e.g. uma Pessoa desempenha um papel de Viajante em uma Viagem); um *<<relator>>* é rígido e conecta outros tipos através de relações *<<mediation>>*, sendo existencialmente dependente das instâncias que ele conecta através de relações *<<mediation>>* (e.g. um acidente de trânsito que faz mediação entre veículos acidentados, uma estrada e vítimas do acidente); um *<<relator>>* também induz

uma relação *<<material>>* entre os tipos mediados (e.g. relação entre um veículo acidentado e a estrada em que o acidente ocorreu); uma classe estereotipada com *<<phase>>* representa um conceito antirrígido e é definido em uma partição de um *<<kind>>* (e.g. uma Pessoa que está viva ou morta, possui como *<<phase>>* os conceitos Pessoa Viva e Pessoa Morta).

2.2 A Linguagem Lógica Alloy

Alloy (JACKSON, 2012) foi criada por Daniel Jackson, é uma linguagem declarativa e baseada na lógica de primeira ordem que descreve e explora estruturas.

Os modelos em Alloy são restrições que descrevem (implicitamente) um conjunto de estruturas. Um modelo é basicamente compreendido em declarações de assinaturas (*signatures*), relações (*relations*), fatos (*facts*), predicados (*predicates*), asserções (*assertions*) e funções (*functions*). A ferramenta Alloy suporta um *solver* que é responsável por encontrar estruturas do modelo que satisfaçam essas restrições. O analisador Alloy (como a ferramenta é chamada) usa os predicados para explorar o modelo gerando exemplos de estruturas em conformidade com o modelo e usa as asserções para checar propriedades do modelo gerando contraexemplos apresentando os resultados graficamente (http://alloy.mit.edu/alloy/). As estruturas em Alloy são compostas de átomos e de relacionamentos entre os átomos:

2.2.1 Átomos e Relações

Um átomo é uma entidade primitiva que é indivisível (não pode ser quebrada em partes menores), imutável (suas propriedades não mudam com o passar do tempo), e não interpretável (não possuem propriedades *built-in* como os números). Poucas coisas no mundo real são atômicas, portanto para criar estruturas que são divisíveis, mutáveis ou interpretáveis, introduzem-se relações para capturar estas propriedades.

Uma relação é uma estrutura que relaciona átomos. Consiste em um conjunto de tuplas, onde cada tupla é uma sequência de átomos. Por exemplo, na Figura 7 temos os átomos

A2 e B4 e uma relação entre eles, a saber, a tupla = (A2, B4). A relação r é então o conjunto de tuplas: r = {(A2, B4), (A1, B4), (A3, B1), (A3, B2), (A3,B3)}.

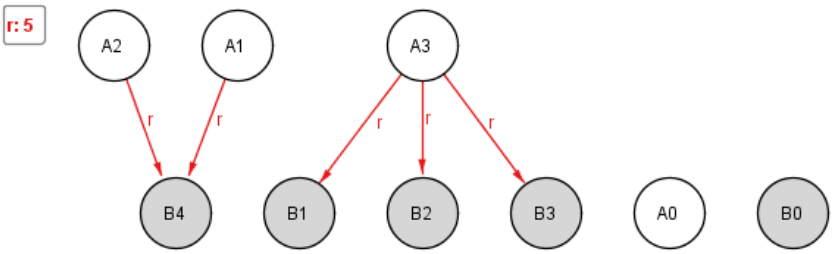

Figura 7. Um exemplo em Alloy.

2.2.2 Assinaturas

Um conjunto de átomos é declarado através da declaração de uma assinatura (*signature)*. Por exemplo, a declaração *sig A {}* declara um conjunto de átomos denominado *A*.

Portanto, uma assinatura é mais que um conjunto de átomos, já que pode incluir também declarações de relações.

As relações são declaradas como campos das assinaturas (*fields* of *signatures)*. Por exemplo, a declaração *sig A {r: set B}* introduz uma relação r cujo domínio é o conjunto A, e cujo contradomínio é o conjunto B, como mostrado na Figura 7.

2.2.3 Fatos, Funções, Predicados e Asserções

As estruturas em Alloy são acompanhadas de propriedades, i.e., restrições e asserções sobre a própria estrutura. Essas propriedades são organizadas em parágrafos. Os parágrafos podem conter quatro tipos de propriedades: fatos (*facts*), funções (*functions*), predicados (*predicates*) ou asserções (*assertions*).

Os fatos (*facts*) especificam restrições que são satisfeitas em qualquer situação. Por exemplo, na Figura 7 poderíamos especificar uma restrição de que o conjunto de átomos A não pode ser um conjunto vazio. Sendo assim, em qualquer possibilidade de instanci-

ação, o conjunto A sempre será um conjunto que possui no mínimo um átomo. Em Alloy, isso seria representado da seguinte forma: *fact {some A}.*

As funções (*functions*) especificam expressões que são empacotadas para serem reusadas no modelo (semelhante ao conceito de funções conhecidas em linguagens de programação). Por exemplo, na Figura 7 poderíamos especificar uma função (chamada, por exemplo, *getBs*) que, dado um átomo do conjunto A, por exemplo, o átomo A1, nos retornasse todos os átomos do conjunto B no qual A1 está relacionado pela relação r. Em Alloy, isso seria representado da seguinte forma: *fun getBs [x: A]: set B {x.r}.*

Os predicados (*predicates*) especificam restrições que podem ser usadas em diferentes contextos, ou seja, em diferentes partes do modelo. Por exemplo, na Figura 7 poderíamos especificar um predicado (chamado, por exemplo, *only2Bs*) com a restrição de que um átomo do conjunto A, por exemplo, o átomo A1, se relaciona com exatamente dois átomos do conjunto B. Em Alloy isso seria representado da seguinte forma: *pred only2Bs [x: A] {# x.B = 2}*. Ainda no que se refere a esse predicado, poderíamos usá-lo para, por exemplo, dizer que todos os átomos do conjunto A do modelo devem satisfazer essa restrição (utilizar o predicado dentro de um fato). Isso seria na seguinte forma: *fact { all x: A | only2Bs[x] }*. Ou seja, a restrição (predicado) em Alloy também pode ser usada em diferentes lugares.

As asserções (*assertions*) especificam propriedades que esperamos que sejam válidas a partir dos fatos do modelo. O analisador Alloy checa as asserções. Se uma asserção não é válida a partir dos fatos, ou uma falha de projeto foi exposta ou houve um erro de formulação da asserção. Por exemplo, poderíamos especificar uma asserção (nomeada *notEmptyBs*) e dizer que o conjunto B sempre será um conjunto não vazio. Isso se daria na seguinte forma: *assert notEmptyBs {some B}*. Se a partir dos fatos do modelo (das restrições que sempre devem ser satisfeitas) o analisador encontrar um exemplo no qual essa asserção não é satisfeita, ele mostrará este caso como um contraexemplo. Caso contrário a asserção pode ser válida ou não.

2.2.4 Comandos e Escopos

Um modelo Alloy constitui-se basicamente de conjuntos e relações, formando a estrutura do modelo, e de fatos, predicados, asserções e funções que especificam as propriedades do mesmo. A ferramenta procura por exemplos e contraexemplos através de comandos, a saber, o comando *run* sobre predicados e o comando *check* sobre asserções.

No caso do predicado (*predicate*), a restrição de análise é a restrição do predicado em conjunto com os fatos do modelo. Uma instância é um exemplo, um cenário no qual ambos, fatos e predicados, são válidos. Por exemplo, a execução do predicado only2Bs da subseção anterior em Alloy seria da seguinte forma: *run only2Bs for 5*. Onde o número 5 especifica o escopo deste comando em Alloy.

No caso da asserção (*assertion*), a restrição de análise é a negação da restrição da asserção em conjunto com os fatos do modelo. Uma instância é um contraexemplo, um cenário no qual a asserção falha a partir dos fatos. Por exemplo, a checagem da asserção da subseção anterior em Alloy seria da seguinte forma: *check notEmptyBs for 5*.

Através da especificação de um escopo, i.e., da quantidade (máxima) de átomos que cada assinatura (*signature*) deve ter em cada cenário (instanciação) do modelo, a ferramenta provê uma possibilidade de instanciação, em forma visual, que satisfaça o modelo.

2.2.5 Análise

Em Alloy é impossível garantir quando uma asserção é válida, pois requer a cobertura de todo o espaço de solução. Ao invés disso, a análise de Alloy é baseada na procura de instâncias (*instance finding)* que é uma tentativa de encontrar uma refutação checando a asserção contra um grande número de casos de testes (um pequeno modelo com apenas quatro relações em Alloy pode ter um espaço de soluções contendo bilhões de casos de teste) (JACKSON, 2012, p.141, 142).

A análise feita pela ferramenta Alloy é baseada na tecnologia SAT (*boolean satisfiability*). O analisador traduz as restrições Alloy para restrições booleanas, as quais são fornecidas a um *solver SAT* eficiente. Este solver pode examinar espaços com uma largura de centenas de bits (ou seja, 10^{60} casos ou mais) (JACKSON, 2012, Prefácio).

A análise de uma asserção inválida termina quando a primeira instância é encontrada. Se nenhuma instância é encontrada, ainda é possível que uma instância exista em um caso de teste maior do que o considerado (JACKSON, 2012, p.141, 142). O tamanho do caso de teste pode ser aumentado mudando-se o valor do escopo (*scope)*. Um escopo em Alloy determina o número máximo de átomos de cada assinatura do modelo (JACKSON, 2012, p.130).

É verdade que a asserção é checada contra um número finito de casos de teste que ocupa apenas uma pequena proporção de todo o espaço de casos possíveis. Entretanto, a análise possui uma cobertura mais ampla do que os testes tradicionais, de modo que tende a ser muito mais eficaz para encontrar problemas de especificação. A busca pela instância que satisfaça a asserção inválida é realizada exaustivamente dentro do pequeno conjunto de casos de teste definidos pelo escopo. A hipótese do escopo pequeno (*small scope hypothesis*) diz que a maioria dos problemas de especificação tem contraexemplos pequenos, ou seja, se uma asserção é inválida, provavelmente se tem um contraexemplo com escopo pequeno dentre todos os casos de testes (JACKSON, 2012, p.143) (ANDONI A. et al., 2002).

2.3 A Transformação de OntoUML para Alloy

A transformação de OntoUML para Alloy tem sido constantemente melhorada desde a sua concepção inicial. Portanto, a especificação Alloy gerada por esta transformação é um pouco diferente daquela apresentada originalmente em (BENEVIDES et al., 2011). Serão abordados aqui apenas os aspectos importantes e relevantes dessa transformação para este projeto.

2.3.1 Estrutura de Mundo Temporal

A transformação OntoUML cria na especificação Alloy resultante uma estrutura de Mundo Temporal (*Temporal World*) adicionada de axiomas da ontologia de fundamentação UFO.

As meta-propriedades de UFO que caracterizam a maioria das distinções ontológicas em OntoUML, são modais por natureza. Alloy não possui essa noção de modalidade, assim, a transformação OntoUML cria na especificação resultante uma estrutura de Mundo Temporal, i.e., uma estrutura com mundos passados, presentes, futuros e contra factuais, que revelam a possível dinâmica de criação, classificação, associação e destruição dos objetos (BENEVIDES et al., 2011, p.3, 19).

A estrutura de Mundo Temporal criada é definida por assinaturas (*signatures*) em Alloy. A estrutura tem a assinatura abstrata *World* como principal assinatura da estrutura (i.e., uma assinatura abstrata não possui elementos exceto aqueles pertencentes às suas extensões). A assinatura *World* (Mundo) é estendida em quatro assinaturas: *CurrentWorld (*Mundo Presente), *PastWorld (*Mundo Passado*), FutureWorld* (Mundo Futuro) e *CounterfactualWorld (*Mundo Contra Factual*)*. Cada Mundo representa um *snapshot* que contém os objetos e relações que existem naquele mundo (BENEVIDES et al.,2011, p.3, 19).

2.3.2 Transformando o Exemplo OntoUML em Alloy

A Figura 8 representa parte do código Alloy gerado pela transformação do modelo OntoUML da Figura 6. (A especificação Alloy completa resultante dessa transformação encontra-se no Anexo A deste projeto).

```
1    ⋮
2    sig Object {}
3    sig Property {}
4    ⋮
5    abstract sig World {
6       exists: some Object+Property,
7       Victim: set exists:>Object,
8       DeceasedPerson: set exists:>Object,
```

```
 9        RoadTrafficAccident: set exists:>Property,
10        ⋮
11        fatalvictims: set RoadTrafficAccident set -> one Int,
12        has: set RoadTrafficAccident one -> some Victim
13    }
14    ⋮
15    fun victims [x: World.RoadTrafficAccident, w: World]: set
16    World.Victim {x.(w.has)}
17    ⋮
```

Figura 8. Transformando o exemplo OntoUML em Alloy.

As classes OntoUML são transformadas em relações binárias entre o mundo (*World*) e o respectivo objeto que existe naquele mundo e que representa instâncias daquelas classes OntoUML. Por exemplo, nas linhas 7,8 e 9, temos uma representação das classes *<<role>> Victim*, *<<phase>> DeceasedPerson* e *<<relator>> RoadTrafficAccident*.

As associações e atributos OntoUML são transformados em relações ternárias entre o mundo e os tipos envolvidos nas associações e atributos em questão. Por exemplo, nas linhas 11 e 12, temos o número de vítimas fatais (o atributo *faltavictims* da classe *<<relator>> RoadTrafficAccident*) e o relacionamento entre acidentes e vítimas (a relação <<mediation>> *has*).

Os tipos de dados definidos em OntoUML (*DataTypes*) são transformados em assinaturas, os finais de associação em OntoUML (*Association Ends*), são transformados em funções em Alloy. Por exemplo, nas linhas 15 e 16, o final de associação *victims* é transformado para função *victims* em Alloy.

2.3.3 Validando o Exemplo OntoUML sem Restrições de Domínio

A validação de modelos conceituais OntoUML via Alloy se dá através da execução da especificação gerada pela transformação OntoUML no analisador Alloy. O analisador então instancia a especificação Alloy encontrando estruturas que satisfaçam as restrições da especificação apresentando-as graficamente.

Ao executar a especificação Alloy do Anexo A no analisador (*Alloy Analyzer*), somos confrontados com a possibilidade de instanciação representada na Figura 9.

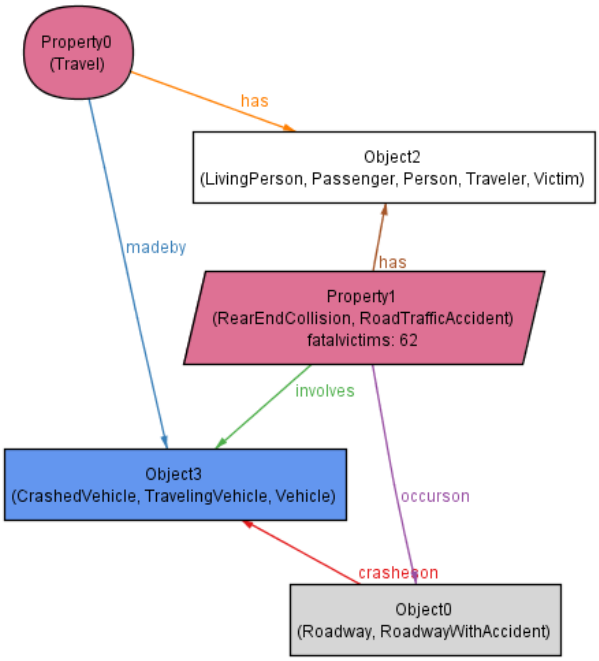

Figura 9. Current World - sem restrições de domínio.

Esta instanciação considera em um mundo atual (*CurrentWorld*), um acidente de trânsito (*Property1*) do tipo colisão traseira (um acidente envolvendo exatamente dois veículos em que um veículo bate na traseira de outro veículo), que ocorre (*occurs on*) em uma estrada (*Object0*), e que possui (*has*) apenas uma vítima (*Object2*), a saber, o passageiro e viajante do veículo (*Object3*) acidentado.

Aqui, somos confrontados com o fato de que o acidente, por ser do tipo colisão traseira, deveria envolver exatamente dois veículos acidentados, mas pelo modelo, é possível que envolva apenas um. Além disso, o número de vítimas fatais é 62 ao invés de nenhum, pois a única vítima do acidente está viva. E mais, o veículo acidentado possui apenas um passageiro e nenhum motorista!

O modelo OntoUML representa o domínio de acidente de trânsito é incapaz de fornecer esses aspectos da conceituação de domínio. É indesejado que um acidente do tipo colisão traseira envolva apenas um veículo acidentado e que o número de vítimas fatais do acidente não seja, por exemplo, derivado do número de vítimas mortas, e também, que haja veículos sem motorista.

Na verdade, as linguagens de modelagem de notação diagramática que são derivados da notação UML, como OntoUML, são incapazes de, representando o domínio de acidente de trânsito, expressar os dois primeiros tipos de restrições (com exceção da restrição de que todo veículo tem um motorista, pois sua representação é possível apenas especificando o modelo OntoUML de forma diferente). Assim, a linguagem de modelagem conceitual OntoUML não possui expressividade o suficiente. É necessária a inclusão de restrições adicionais no modelo para uma especificação mais próxima da conceituação de domínio.

De acordo com esse snapshot e o que foi dito até aqui, três restrições tornam-se necessárias:

(a) Todo acidente do tipo colisão traseira deve envolver exatamente dois veículos acidentados.

(b) O número de vítimas fatais de um acidente deve ser derivado do número de vítimas mortas no acidente de trânsito.

(c) Todo veículo que faz uma viagem deve possuir exatamente um motorista.

Como discutido anteriormente, essas restrições podem ser escritas na linguagem de restrições de objetos chamada OCL, que será discutida no próximo capítulo. As restrições serão agregadas ao modelo OntoUML e serão objeto de transformação para Alloy, como discutido no capítulo 4.

3 A Linguagem de Restrições de Objetos OCL

Este capítulo introduz OCL e o subconjunto de restrições considerado através da definição de restrições no exemplo OntoUML. As restrições OCL escritas aqui são aquelas descobertas na subseção 2.3.3 por meio da validação do exemplo OntoUML sem restrições de domínio.

A seção 3.1 introduz um pequeno histórico da linguagem. A seção 3.2 apresenta OCL definindo seu escopo de utilização, ou seja, estabelecendo o subconjunto de restrições OCL considerado neste projeto, i.e., invariantes na seção 3.2.1 e derivações na seção 3.2.2.

3.1 Breve Histórico

OCL é um padrão do grupo OMG (*Object Management Group*). Foi inicialmente desenvolvida pela IBM (*International Business Machines*). Inicialmente era apenas uma extensão da linguagem de modelagem unificada UML para descrever restrições sobre a mesma. Atualmente, um subconjunto da linguagem OCL pode ser usado com qualquer metamodelo MOF (a linguagem completa só pode ser usada com a linguagem UML) (OMG, 2012, OCL 2.3, p.1).

A partir da especificação OCL 2.x, a linguagem não só é usada como uma linguagem de restrições, mas também como uma linguagem de consulta (OMG, 2012 OCL 2.3, p.8).

Além disso, OCL desempenha um papel importante na especificação QVT (*Query/View/Transformation*), outro padrão do grupo OMG, que especifica um conjunto de linguagens para transformações de modelos.

Portanto, a linguagem OCL hoje é utilizada de diversas formas: para criar modelos, criar linguagens e transformar modelos. A linguagem OCL será tratada aqui como uma linguagem de especificação de restrições em diagramas de classe da UML (e assim, por consequência, também em diagramas OntoUML).

É importante ressaltar que a linguagem OCL neste trabalho não é utilizada como uma linguagem de restrições ou consultas sobre metamodelos (i.e., modelos no nível M2 da hierarquia da *Model-Driven Architecture*), e sim, como uma linguagem de restrições sobre modelos (i.e., modelos no nível M1), a saber, modelos OntoUML.

3.2 A Linguagem

OCL é uma linguagem textual formal (o grau de formalidade de OCL está em discussão, mas é considerada ao menos uma linguagem semiformal) (CABOT; GOGOLLA, 2012, p.60), adotada como um padrão da OMG para representar restrições em modelos baseados em MOF.

A linguagem OCL é baseada na teoria dos conjuntos e na lógica de predicados. Sua notação não usa símbolos matemáticos. O argumento dos projetistas da linguagem baseia-se na observação de que a experiência com notações formais ou matemáticas leva a seguinte conclusão: as pessoas que usam a notação matemática conseguem expressar coisas precisas e sem ambiguidade, porém poucas pessoas conseguem realmente entender tais notações (WARMER; KLEPPE, 2003).

É uma linguagem dita *tipada* onde cada expressão OCL possui um tipo e onde, para ser bem formada, cada expressão precisa estar em conformidade com as regras de tipos da linguagem (as classes de um modelo UML fazem parte dos tipos da linguagem OCL).

OCL é também uma linguagem declarativa, e sem efeitos colaterais, ou seja, quando uma expressão OCL é avaliada, ela simplesmente retorna um valor, e o estado do sistema não sofre alteração por causa da avaliação dessa expressão OCL.

A linguagem OCL é utilizada para diversos propósitos (OMG 2012, OCL 2.3, p.5,6), incluindo a especificação de:

- Invariantes sobre os tipos do modelo (i.e., restrições de integridade);

- Derivações para finais de associações (*association ends*) e atributos;
- Pré-condições e pós-condições sobre operações e métodos;
- Operações de consulta; e,
- Valores iniciais para atributos.

Com o propósito de complementar o modelo OntoUML com restrições a fim de capturar acuradamente a conceituação, serão tratadas aqui invariantes e derivações OCL. As invariantes especificam todas as condições necessárias que devem ser satisfeitas em cada instanciação possível do modelo. As derivações por sua vez expressam como os valores derivados dos elementos do modelo devem ser computados. Restrições que envolvem operações (pré-condições, pós-condições, etc.) e valores iniciais de atributos estão fora do escopo deste trabalho, pois estes elementos não fazem parte de OntoUML.

3.2.1 Invariantes

Invariantes são condições booleanas aplicadas a Tipos (*Types*) do modelo, i.e., Classes ou Tipos de dados (*DataTypes*). A condição deve ser verdadeira para cada instância do tipo em qualquer ponto no tempo. A Figura 10 especifica duas invariantes.

```
context RearEndCollision inv:
self.vehicles->size()=2

context TravelingVehicle inv:
self.travel.travelers->one(t | t.oclIsKindOf(Driver))
```

Figura 10. Invariantes OCL sobre o exemplo OntoUML.

A primeira invariante (a) especifica uma condição booleana (a expressão depois da palavra-chave *inv*) sobre o Tipo *RearEndCollision (<<relator>>)* do modelo. O Tipo é conhecido como o contexto da invariante. Essa restrição especifica que cada acidente de trânsito que é do tipo colisão traseira (*<<relator>> RearEndCollision*) deve envolver exatamente dois veículos, em qualquer ponto no tempo.

A variável *self* representa uma instância arbitrária do contexto, ou seja, uma instância do *<<relator>> RearEndCollision*. A notação *dot* (,) de OCL é usada tanto para acessar atributos ou finais de associação como para a chamada de métodos. Assim, a expressão *self.vehicles* retorna um conjunto de veículos do tipo *<<role>> CrashedVehicle* que estão envolvidos no acidente de colisão traseira *self*. Por último, a operação OCL *size* retorna o número de elementos desse conjunto de veículos.

Note que a condição deve ser satisfeita não apenas para a instância *self*, mas para todas as instâncias do contexto, ou seja, para todo *<<relator>> RearEndCollision*.

A segunda invariante (b) possui como contexto veículos que fazem viagem *<<role>> TravelingVehicle).* A expressão *self.travel.travelers* refere-se ao conjunto de viajantes da viagem feita pelo veículo *self*. O iterador OCL *one* retorna verdadeiro se de todos os viajantes, exatamente um deles for um motorista (*Driver*).

3.2.2 Derivações

Derivações expressam como um valor derivado pode ser inferido a partir de outros elementos do modelo (CABOT; GOGOLLA, 2012, p.62). A Figura 11 especifica que o número de vítimas fatais de acidentes de trânsitos é derivado da soma do número de vítimas mortas em cada acidente.

```
context RoadTrafficAccident :: fatalvictims : int
derive: self.victims->select(d| d.oclIsKindOf(DeceasedPerson))->size()
```

Figura 11. Derivação OCL sobre o exemplo OntoUML.

O contexto da derivação é o atributo *fatalvictims* do *<<relator>> RoadTrafficAccident* (do tipo primitivo *int*). O iterador OCL *select* retorna o conjunto dos elementos que satisfazem a condição dentre parênteses, ou seja, retorna o conjunto de vítimas (*<<role>> Victim*) que estão mortas (instâncias também de *<<phase>> DeceasedPerson*).

Note que a derivação não especifica que o valor do elemento derivado não pode mudar, ela apenas diz que ele sempre irá mudar de acordo com a avaliação da derivação (CABOT; GOGOLLA, 2012, p.62).

4 A Transformação de OCL para Alloy

Este capítulo apresenta a transformação de restrições OCL para Alloy compatível com a transformação OntoUML abordada na seção 2.3.

A seção 4.1 apresenta uma visão geral da transformação. A seção 4.2 por sua vez descreve todos os mapeamentos da linguagem OCL para Alloy suportados neste projeto. A seção 4.3 apresenta a validação do exemplo OntoUML com as restrições de domínio OCL definidas nas subseções 3.2.1 e 3.2.2, transformando essas restrições OCL para Alloy através dos mapeamentos definidos na subseção 4.2.

4.1 Visão Geral

Fatos em Alloy são restrições que sempre devem ser satisfeitas (JACKSON, 2012, p.119). Transformar restrições OCL em fatos significa que as instâncias providas pelo analisador Alloy sempre estarão em conformidade com essas restrições.

Restrições OCL são definidas por uma expressão OCL (*OCLExpression*) aplicada em um contexto, como mostrado na Figura 12.

```
context Classifier
inv invariant-name: OCLExpression

context Classifier::Attribute:Type
derive: OCLExpression
```

Figura 12. Formato de uma invariante e uma derivação OCL.

No caso de invariantes esse contexto é uma classe OntoUML. Em derivações, o contexto é um final de associação (*Association End*) ou um atributo de uma classe.

Expressões OCL devem ser verdadeiras para toda instância do contexto em todo Mundo Temporal, seja mundo Presente, Futuro, Passado ou Contra Factual.

No caso de invariantes, podemos dizer que, para todo mundo *w* e para toda instância *self* do contexto contida no mundo *w*, a expressão *OCLExpression* deve ser verdadeira. No caso de derivações, podemos dizer que, para todo mundo *w* e para toda instância *self* do contexto contida no mundo *w***,** o atributo da instância *self* deve sempre ser igual à avaliação da expressão *OCLExpression*.

A Figura 13 mostra o formato dos fatos correspondentes a invariantes e derivações, considerando-se a estrutura de mundos representada em Alloy. A notação *TR(x)* indica a transformação de *x* para Alloy.

```
1
2    fact TR(invariant-name) {
3      all w: World | all self: w.TR(Classifier) |
4      TR(OCLExpression)
5    }
6
7
8    fact {
9      all w: World | all self: w.TR(Classifier) |
10     self.(w.TR(Attribute)) = TR(OCLExpression)
11   }
```

Figura 13. Formato de uma invariante e uma derivação em Alloy.

Os nomes das invariantes (*TR(invariant-name)*) necessitam ser transformados em Alloy pra evitar conflitos de nomes. O contexto da invariante ou da derivação (*TR(Classifier)*) necessita ser transformado para sua referência correta em Alloy de acordo com a transformação OntoUML. Da mesma forma, os atributos a serem derivados também são transformados de acordo com a transformação OntoUML (*TR(Attribute)*). E finalmente, as condições que devem ser satisfeitas, i.e., as expressões OCL (*OCLExpressions*) também necessitam ser transformadas para Alloy.

Uma expressão OCL (*TR(OCLExpression)*) é formada por outras expressões OCL e por tipos OCL e suas operações. Assim, na próxima subseção, descreveremos todos os mapeamentos da linguagem OCL suportados, a saber: tipos e suas operações, expressões e restrições.

4.2 Os Mapeamentos

OCL é definida por Tipos (4.2.1) e Expressões (4.2.2) (OMG, 2012, OCL 2.3, p.35), as quais são usadas para a construção das restrições, i.e., para determinar as condições que devem ser satisfeitas. Esta seção apresenta a transformação dos tipos, suas operações, de expressões e de restrições OCL para a linguagem Alloy.

4.2.1 Tipos e suas Operações

A Figura 14 representa um fragmento do metamodelo de Tipos OCL, adaptado de (OMG, 2012 OCL 2.3.1 p.36).

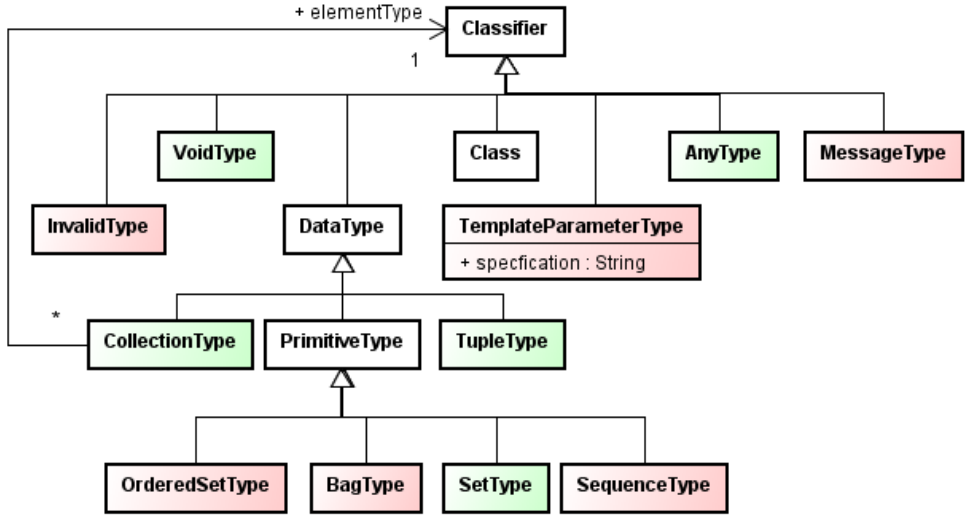

Figura 14. Fragmento do metamodelo de Tipos OCL.

As metaclasses de cor branco são provenientes do metamodelo de UML. As metaclasses de cor verde claro definem os tipos OCL que são suportados pela nossa transformação. Portanto, nas subseções seguintes serão abordados os mapeamentos dos Tipos de OCL (Any, Void e Coleções) e Tipos do Modelo (incluindo tipos primitivos), i.e., classes de cor verde claro e de cor branco, e suas operações para a linguagem Alloy.

Já as metaclasses de cor vermelho claro definem os tipos OCL que não são suportados em nossos mapeamentos. Alguns devidos a insignificância à OntoUML, como por exemplo as metaclasses *MessageType* e *TemplateParameterType* (OntoUML não possui

o conceito de mensagens e parâmetros de template) e as metaclasses *OrderedSetType* e *SequenceType* (OntoUML não possui os conceitos de conjuntos ordenados e sequências), outros pela incompatibilidade com a linguagem Alloy, como a metaclasse *InvalidType* (Alloy não possui um valor inválido para erros de conformidade de tipos), e finalmente pela incompatibilidade com a transformação de Benevides (2011) e a linguagem Alloy, como a metaclasse *BagType* (não é possível representar coleções do tipo Bag em Alloy de acordo com a transformação de OntoUML para Alloy considerada).

4.2.1.1 Tipos Especiais

OCL possui como tipos especiais *OclAny* (instância *singleton* da metaclasse *AnyType*), *OclVoid* (instância *singleton* da metaclasse *VoidType*), *OclInvalid* (instância *singleton* da metaclasse *InvalidType*). Alloy não suporta alguns elementos de OCL como, por exemplo, o tipo inválido (OclInvalid), pois Alloy não é uma linguagem tipada e baseia-se na noção de conjuntos. Assim, com relação aos tipos especiais, Alloy suporta somente os tipos *OclAny* e *OclVoid*.

OclAny é o supertipo para todos os tipos (com exceção dos tipos de coleção OCL) (OMG, 2012, OCL 2.3, A.4.6). Em Alloy, a constante *univ* desempenha o papel de super conjunto ao qual engloba todos os conjuntos do modelo (JACKSON, 2012, p.50). *OclAny* pode ser representado em Alloy pela expressão *univ*-*World* (a constante *univ* excluindo o conjunto *World*).

OclVoid, por sua vez, possui um único valor indicando a ausência de valor, a saber, o valor *null* (OMG, 2012, OCL 2.3, A.4.6). Em Alloy, a constante *none* desempenha o papel de um subconjunto de todos os outros conjuntos do modelo (semelhante ao conjunto vazio de teoria dos conjuntos) (JACKSON, 2012, p.50). *OclVoid* pode ser representado em Alloy pela constante *none*, i.e., conjunto vazio em Alloy.

A Tabela 1 mostra as operações suportadas para *OclAny*. Note que cada operação OCL tem um argumento implícito pela chamada da operação (i.e., *self*). A notação *TR(x)* indica a transformação de *x* para Alloy.

OCL	Alloy
oclIsKindOf(*t*: Classifier): Boolean	TR(*self*) in TR(*t*)
oclIsTypeOf(*t*: Classifier): Boolean	TR(*self*) in TR(*t*) and # TR(*self*) & (TR(*subtype*)+…+TR(*subtype*)) = 0
oclIsUndefined(): Boolean	TR(*self*) = none
oclAsType(*t*: Classifier): Classifier-instance	TR(*self*)
= (*obj*: OclAny): Boolean	TR(*self*) = TR(*obj*)
<> (*obj*: OclAny): Boolean	TR(*self*) != TR(*obj*)

Tabela 1. Mapeamento das operações de tipos especiais OCL para Alloy.

A operação *oclIsKindOf* é representada em Alloy pelo operador *in*. Note que o parâmetro implícito *self* e o parâmetro *t* são também transformados para Alloy (*TR(self)* e *TR(t)*).

A operação *oclIsTypeOf* é representada em Alloy através de vários operadores Alloy, i.e., *in*, *and*, *#, &,* + e =. Esta operação significa que *self* é do tipo *Classifier* e não de um subtipo de *Classifier*. A parte da expressão Alloy resultante *TR(subtype)+…+TR(subtype)* representa a união de todos os subtipos do classificador *t*.

Além disso, no contexto de expressões de navegação, nós assumimos o mesmo que (ANASTASAKIS, 2008) representando o valor indefinido OCL (OCL *undefined value*) como o conjunto vazio, porque no contexto de avaliações de erros, o valor indefinido pode significar também o valor OCL inválido (*OclInvalid*), ao qual Alloy não suporta.

A operação *oclAsType* é representada em Alloy transformando apenas o parâmetro *self* (*TR(self)*) em Alloy, pois Alloy não possui um operador de conformidade de tipos para lidar com erros de conformidade de tipos (*type conformance error*) ou como um valor OCL inválido (resultante de um erro de tipos). Além disso, supondo um erro de conformidade de tipos em Alloy, ele seria capturado na execução da especificação Alloy ao qual seria impedida de prosseguir.

Finalmente, as operações OCL = e <> são representadas em Alloy por seus operadores equivalentes = e *!=*, respectivamente.

4.2.1.2 Tipos do Modelo

De acordo com a transformação OntoUML, a qual é base para nossos mapeamentos, as Classes são representadas em Alloy por relações binárias (tendo o Mundo como domínio e os objetos que representam a classe como imagem) e os Tipos de Dados (*DataTypes*) são representados como assinaturas (*signatures*).

Portanto, classes referenciadas em OCL são transformadas para a expressão Alloy *w.TR(Class)* e tipos de dados referenciados em OCL são transformados para *TR(DataType)* em Alloy. A variável w representa uma instância de Mundo (*World*) na qual instâncias da Classe OntoUML podem existir.

A Tabela 2 apresenta a operação *allInstances* que é suportada para todas as Classes e Tipos de Dados (*DataTypes*) do modelo. A operação é representada em Alloy transformando apenas o parâmetro *self* (*TR(self)*). Essa operação não precisa de nenhum operador equivalente em Alloy, pois Alloy é baseada na noção de conjuntos, e a expressão correspondente a *self* neste caso será um conjunto com todas as instâncias do tipo sendo transformado.

OCL	Alloy
allInstances()	TR(*self*)

Tabela 2. Mapeamento das operações de tipos do modelo para Alloy.

4.2.1.3 Tipos Primitivos

Os tipos primitivos (provenientes da linguagem de modelagem) são: *Integer*, *Boolean*, *Real*, *String* e *UnlimitedNatural*, entretanto, Alloy nativamente somente suporta um subconjunto de Inteiros (*Integer*), a saber, o tipo Alloy *Int*, e o tipo Booleano (*Boolean*) no contexto de expressões ou fórmulas booleanas em Alloy. A Tabela 3 mostra as operações suportadas para os tipos primitivos .

OCL	Alloy
and (*b*: Boolean) : Boolean	TR(*self*) and TR(*b*)
or (*b*: Boolean) : Boolean	TR(*self*) or TR(*b*)
implies (*b*: Boolean) : Boolean	TR(*self*) implies TR(*b*)
not : Boolean	not TR(*self*)
xor (*b*: Boolean) : Boolean	(TR(*self*) or TR(*b*)) and not (TR(*self*) and TR(*b*))
> (*i*: Integer) : Boolean	TR(*self*) > TR(*i*)
< (*i*: Integer) : Boolean	TR(*self*) < TR(*i*)
>= (*i*: Integer) : Boolean	TR(*self*) >= TR(*i*)
<= (*i*: Integer) : Boolean	TR(*self*) <= TR(*i*)
max (*i*: Integer) : Boolean	max[TR(*self*),TR(*i*)]
min (*i*: Integer) : Boolean	min[TR(*self*),TR(*i*)]
abs () : Integer	abs[TR(*self*)]
- : Integer	negate[TR(*self*)]
+ (*i*: Integer) : Integer	TR(*self*).plus[TR(*i*)]
- (*i*: Integer) : Integer	TR(*self*).minus[TR(*i*)]
* (*i*: Integer) : Integer	TR(self).mul[TR(*i*)]

Tabela 3. Mapeamento de operações de tipos primitivos para Alloy.

As operações booleanas *and, or, implies,* e *not*, e as operações inteiras >, <, >= e <=, são representadas em Alloy por seus operadores equivalentes. A operação *xor* é representada em Alloy indiretamente através dos operadores *and, or,* e *not*, pois o operador *xor* não existe em Alloy. As operações aritméticas como - (negação), + (adição), - (subtração) e * (multiplicação) são representadas em Alloy através das funções fornecidas pela biblioteca inteira de Alloy, i.e., *negate[], plus[], minus[]* e *mul[],* respectivamente.

As operações *max, min* e *abs,* foram implementadas em Alloy pois o módulo inteiro de Alloy não especifica essas operações. A Tabela 4 mostra essas operações aritméticas implementadas.

OCL	Alloy
abs () : Integer	fun abs [self: Int] : Int

	{ self < 0 implies self.negate else self }
min (i: Integer) : Integer	fun min [self, i: Int] : Int
	{ let a = int[self], b = int[i] | a <= b implies a else b }
max (i: Integer) : Integer	fun max [self, i: Int] : Int
	{ let a = int[self], b = int[i] | a <= b implies b else a }

Tabela 4. Implementação de operações aritméticas em Alloy.

Alloy possui um valor primitivo booleano (*PrimitiveBoolean*) que é retornado em cada avaliação de uma fórmula. Este valor não pode ser declarado, ele é apenas um resultado de uma fórmula em Alloy. Assim, toda fórmula em Alloy resulta em *PrimitiveBoolean*, como é o caso das expressões booleanas com as operações *and, or, implies, xor* e *not,* as operações aritméticas de comparação >, <=, *e* etc. Ou seja, *PrimitiveBoolean* é o tipo resultante de todos os mapeamentos que resultam em um valor booleano.

Alloy possui um tipo *Bool* fornecido pela biblioteca booleana de Alloy, útil quando o tipo *Boolean* precisa ser declarado (JACKSON, 2012, p.137, 138, 139), por exemplo, quando é necessário modelar valores literais como *true* ou *false* explicitamente (o que não é possível com o *PrimitiveBoolean).* Esse tipo *Bool* é implementado como uma assinatura em Alloy (*sig Bool {}*) e possui como sub-assinaturas os conjuntos *True* e *False (one sig True, False extends Bool {}).*

Declarando um atributo em um modelo do tipo *Boolean (*tipo primitivo do modelo*)*, cria em Alloy uma relação com essa assinatura *Bool,* e especificando literais *true* ou *false* a esse atributo cria em Alloy relações com essas sub-assinaturas *False* ou *True*.

Por exemplo, uma classe <<*relator*>> *Meeting* com um atributo booleano *isConfirmed* em um modelo OntoUML cria em Alloy uma relação ternária (World,Meeting,Bool), onde o valor *Bool* pode ser a assinatura *True* ou *False* em Alloy.

Supondo a expressão booleana OCL "*meeting.isConfirmed or true*", sabemos que a operação *or* em Alloy recebe como parâmetros tipos *PrimitiveBoolean,* porém, o atributo *isConfirmed* e o literal *true* são transformados para a assinatura *Bool* em Alloy.

Para resolver essa incompatibilidade de tipos booleanos em Alloy, a biblioteca do tipo *Bool* fornece um predicado, a saber, *isTrue[]* e *isFalse[]*, responsável por avaliar esse valor *Bool* baseado em uma fórmula Alloy e que resulta em um valor primitivo booleano (*PrimitiveBoolean*). Assim, poderíamos escrever o mapeamento da expressão OCL em Alloy como:

" *isTrue[((w.meeting).(w.isConfirmed)] or isTrue[True]* "

Entretanto, embora seja possível transformar o valor *Bool* de Alloy para o valor *PrimitiveBoolean*, a tarefa de reconhecer e estabelecer padrões de quando cada mapeamento deve ser usado em expressões OCL não é simples devido às inúmeras possibilidades de escrita de expressões OCL. Por exemplo, poderíamos especificar alternativamente em OCL *"meeting.isConfirmed = true or true"*. Nesse caso, o primeiro literal booleano seria um *Bool* e o segundo um *PrimitiveBoolean*.

Assim, em nossa proposta de mapeamento, adotamos uma abordagem mais simplista, que não suporta expressões booleanas literais (*true* ou *false*) ou expressões que resultam em um valor booleano de um atributo do modelo, como o exemplo supracitado.

Além disso, o subconjunto de Inteiros suportados é limitado pelo valor da variável *bitwidth* em Alloy. Este valor está compreendido entre [-64,63] para um bom desempenho na análise, podendo ser facilmente incrementado se necessário. Isto significa que as restrições OCL devem considerar a *priori* valores inteiros que estão nesse intervalo.

4.2.1.4 Tipos Coleções

Os tipos coleções de OCL são divididos em: *Set*, *OrderedSet*, *Bag* and *Sequence* (OMG, 2012, OCL 2.3, p.155, 156). Um *Set* em OCL é uma coleção sem elementos duplicados e sem nenhuma ordem. Já um *Bag* em OCL é uma coleção que permite um elemento aparecer mais de uma vez, mas também sem nenhuma ordem. Por sua vez, um *OrderedSet* em OCL é uma coleção sem elementos duplicados mas que possui uma ordena-

ção e um *Sequence* em OCL é uma coleção que permite duplicações e que também possui ordenação.

Nativamente, Alloy suporta somente o tipo coleção *Set*, mas a biblioteca de Alloy fornece um módulo que especifica um tipo *Sequence* (sequência) em Alloy. Em (ANATASAKIS, 2008, p.75, 76), foi mostrado que é possível definir *Bags* e *OrderedSets* a partir do tipo *Sequence* definido em Alloy.

Entretanto, a transformação de OntoUML para Alloy não leva isto em consideração, OntoUML não possui conceitos como coleções ordenadas ou sequências, mas nela é possível a ocorrência de coleções do tipo *Bag*, como por exemplo, no caso de relações materiais derivadas. Apesar disso, a transformação de OntoUML para Alloy transforma todas as classes e tipos de dados do modelo OntoUML para conjuntos (*Sets*) por padrão. Assim, como nossa abordagem é baseada na transformação de OntoUML para Alloy e a complementa, consideramos apenas coleções do tipo *Set*. Isso significa que não é possível escrever restrições OCL em que o tipo *Bag* ocorre no caso específico de relações materiais derivadas em OntoUML. Nesse contexto, só é possível escrever restrições OCL que envolvam *Sets*.

Além disso, Alloy também não suporta coleções aninhadas (*nested collections*), pois não é possível representar *high order relations* em Alloy (JACKSON, 2012, p.41). Uma coleção OCL é parametrizada com um tipo T (*Collection(T)*), por exemplo, a coleção *Set(Person)*. Em OCL, o tipo T pode ser também uma coleção OCL. Assim, coleções dentro de coleções (em nosso caso, conjuntos dentro de conjuntos, e.g., *Set(Set(Person))*) não são suportados em Alloy. Com isso, expressões com o iterador de coleções *collectNested* não são suportados. Isso quer dizer que há um mapeamento parcial da metaclasse *CollectionType* do fragmento do metamodelo de Tipos OCL da Figura 14, pois em Alloy não há o suporte de coleções aninhadas e devido a insignificância de coleções aninhadas à OntoUML.

A Tabela 5 mostra as operações suportadas para conjuntos em OCL (*Sets*). Cada *Set* tem um parâmetro T que denota o tipo (*Type*) do conjunto, por exemplo: *Set(Integer)*,

Set(Person), e etc. Quase todas as operações possuem o mesmo mapeamento de (ANASTASAKIS, 2008, Table2), exceto a operação OCL *excludesAll* que possui aqui uma proposta de mapeamento diferente da de Anastasakis, e a operação OCL *"-"* (diferença de conjuntos) que não foi considerada por Anastasakis pela ambiguidade do operador (porque o mesmo operador é também utilizado para a operação de subtração e negação de inteiros), mas que consideramos neste trabalho.

OCL	Alloy
size() : Integer	# TR(*self*)
includes(*obj*: T) : Boolean	TR(*obj*) in TR(*self*)
includesAll(*s*: Set(T)):Boolean	TR(*s*) in TR(*self*)
excludes(*obj*: T) : Boolean	TR(*obj*) not in TR(*self*)
excludesAll(*s*: Set(T)): Boolean	# (TR(*s*) & TR(*self*)) = 0
isEmpty() : Boolean	no TR(*self*)
notEmpty() : Boolean	some TR(*self*)
union (*s*: Set(T)) : Set(T)	TR(*self*) + TR(*s*)
intersection (*s*: Set(T)) : Set(T)	TR(*self*) & TR(*s*)
- (*s*: Set(T)) : Set(T)	TR(*self*) - TR(*s*)
including(*obj*: T) : Set(T)	TR(*self*) + TR(*obj*)
excluding(*obj*: T) : Set(T)	TR(*self*) – TR(*obj*)
symmetricDifference(*s*: Set(T)) : Set(T)	(TR(*self*) + TR(*s*)) – (TR(*self*) & TR(*s*))
asSet() : Set(T)	TR(*self*)
product(*s*: Set(T2)) : Set(Tuple(first: T, second:T2))	TR(*self*) → TR(*s*)
sum(): T	sum TR(*self*)

Tabela 5. Mapeamento das operações de tipos coleções OCL para Alloy.

A expressão OCL "*Student->excludesAll(Customer)"* especifica que o conjunto *Student* (estudante) não contém nenhum elemento do conjunto *Customer* (cliente), ou seja, o conjunto de estudantes não possue nenhuma interseção com o conjunto de clientes (OMG, 2012, OCL 2.3.1 p.157).

A operação *excludesAll* foi representada por Anastasakis (2008) através do operador Alloy *!in* (não é um subconjunto de). A expressão *"Customer !in Student"* em Alloy não possui o mesmo significado da operação OCL *excludesAll*.

A expressão com o operador *!in* em Alloy especifica que o conjunto *Customer* (cliente) não está contido no conjunto de *Student* (estudante), ou seja, no caso em que há interseção entre os conjuntos *Customer* e *Student* a expressão Alloy é verdadeira.

Assim, a operação OCL *excludesAll é* representada aqui verificando se a interseção dos conjuntos é vazia, ao contrário de Anastasakis que utiliza o operador Alloy *!in*.

4.2.2 Expressões e Restrições

A Figura 15 representa um fragmento do metamodelo de Expressões OCL, adaptado de (OMG, 2012, OCL 2.3.1 p.42).

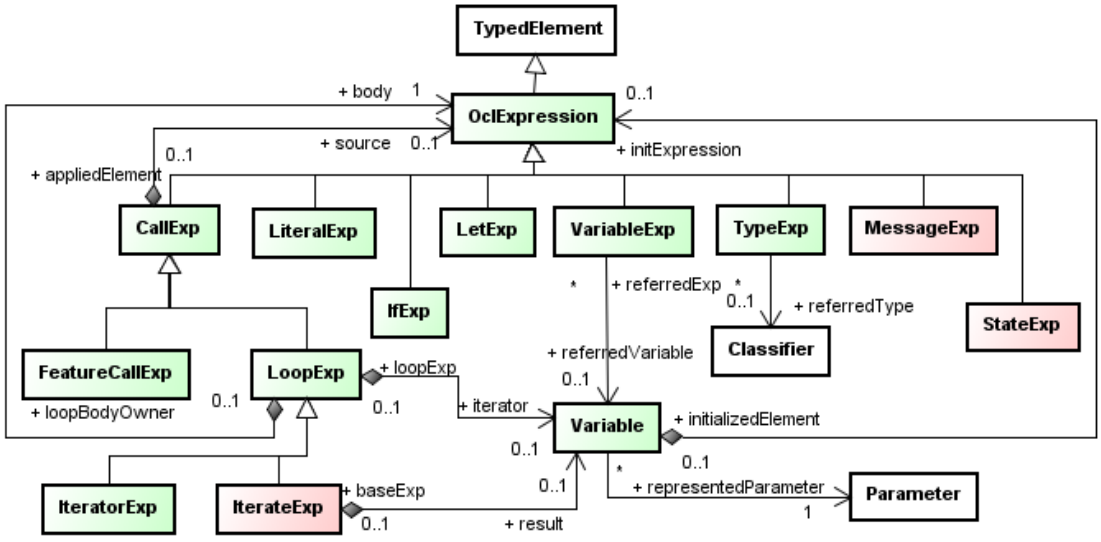

Figura 15. Fragmento do metamodelo de Expressões OCL.

As metaclasses de cor branco são provenientes do metamodelo de UML. As metaclasses de cor verde claro definem os tipos OCL que são suportados pela nossa transformação (além dos tipos provenientes do modelo UML supracitado) e as metaclasses de cor vermelho as que não são suportadas.

Portanto, nas subseções seguintes serão abordados os mapeamentos das Expressões OCL como expressões *If* e *Let*, chamada de propriedades (*FeatureCall)*, expressões de iteradores (*IteratorExp*), literais (*LiteralExp*), e expressões de tipos e variáveis (*TypeExp* e *VariableExp*) para Alloy i.e., classes de cor verde claro e de cor branco, e suas operações para a linguagem Alloy.

Já as metaclasses de cor vermelho claro definem os tipos OCL que não são suportados em nossos mapeamentos, quer seja pela insignificância à OntoUML, como as metaclasses *MessageExp* e *StateExp* (OntoUML não possui o conceito de mensagens e estados), quer seja pela incompatibilidade com a linguagem Alloy, como a metaclasse *IterateExp* (Alloy não possui um construtor capaz de acumular valores sobre um conjunto). Com isso, expressões OCL que possuam a operação OCL *iterate* não são suportados (subseção 4.2.2.2). Embora essa operação não seja suportada, nossos mapeamentos suportam quase todos os iteradores OCL (iteradores OCL são especificados pela linguagem OCL usando a operação *iterate)*.

4.2.2.1 Expressões Predefinidas

A Figura 16 representa um fragmento do metamodelo de Expressões If e Let em OCL, adaptado de (OMG, 2012 OCL 2.3.1 p. 47, 53).

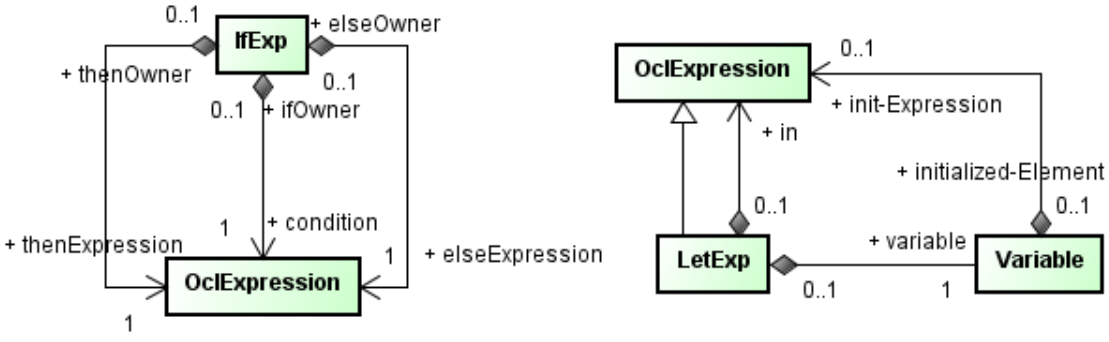

Figura 16. Fragmento do metamodelo de Expressões OCL If e Let.

A Tabela 6 mostra as expressões *if-then-else* e *let-in* representadas em Alloy pelas expressões equivalentes *implies-else* e *let*, respectivamente, onde a variável *be* (*boolean*

expression) é representada em Alloy pela transformação *TR(be)* e a expressão *expr* pela transformação *TR(expr).*

OCL	Alloy
if *be* then *be1* else *be2*	TR(*be*) implies TR(*be1*) else TR(*be2*)
let x : Type = *expr* in *be*	let x = TR(*expr*) | TR(*be*)
self.AssocEnd	TR(*self*).TR(*AssocEnd*)[w]
self.Attribute	TR(*self*).(w.TR(*Attribute*))

Tabela 6. Mapeamento das expressões predefinidas OCL para Alloy.

A Figura 17 representa um fragmento do metamodelo de Expressões de chamada de operações e de navegação OCL, adaptado de (OMG, 2012 OCL2.3.1 p.46).

O acesso de atributos e finais de associação do modelo é o que chamamos de navegação OCL. São representados em Alloy usando a notação *dot*. Embora representados pela mesma notação em Alloy, o acesso a atributos e finais de associação não são mapeados diretamente, pois deve levar em consideração a forma sintática em que os atributos e finais de associação foram mapeados em Alloy pela transformação OntoUML.

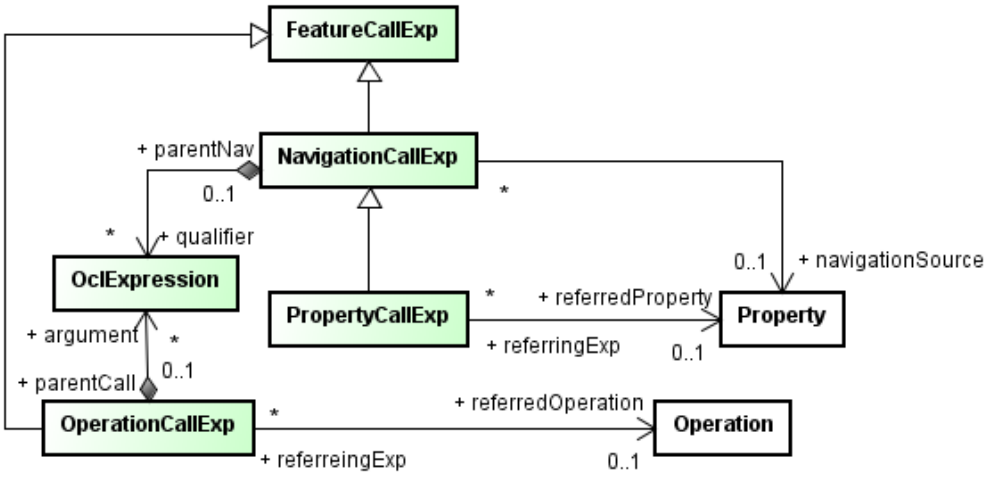

Figura 17. Fragmento do metamodelo de chamada de operação e navegação OCL.

De acordo com a transformação de OntoUML para Alloy, o final de associação (*AssocEnd*) é transformado em uma função em Alloy. Essa função recebe como parâmetros uma instância *w* representando o Mundo (*World*) na qual aquele final de associação pode existir e o tipo *source* (*self*) da associação. Portanto, o acesso a um final de associação do modelo OntoUML é representado em Alloy através da expressão *TR(self).TR(AssocEnd)[w]* (notação alternativa para a chamada de função).

Com relação ao atributo, a transformação de OntoUML para Alloy o representa em Alloy como uma relação ternária entre o Mundo (*World*) e os tipos envolvidos, a saber o *owner* do atributo e o próprio atributo. Portanto, em Alloy, um acesso a um atributo do modelo OntoUML é representado pela expressão *w.TR(Attribute)*.

Como OntoUML não possui conceitos de operações, as chamadas de operações (representada pela metaclasse *OperationCallExp*) são as próprias operações OCL suportadas pelo nosso mapeamento, ou seja, as operações das tabelas dos tipos especiais, tipos primitivos, tipos do modelo, de coleções e etc.

4.2.2.2 Expressões de Iteradores

Iteradores em OCL são implementados através da operação OCL *iterate*, ao qual Alloy não suporta. Entretanto, Alloy possui um mapeamento para quase todos os as expressões de iteradores predefinidos por OCL.

A Tabela 7 mostra todos os iteradores suportados. Note que os iteradores sempre possuem como *source* expressões que resultam em coleções. A expressão coleção *col* e a expressão booleana *be* são representadas por suas respectivas transformações (*TR(col)* e *TR(be)*). Os iteradores OCL *forAll* e *exists* são representados em Alloy por fórmulas usando os operadores *all* e *some*, respectivamente.

Além disso, os iteradores OCL *select*, *reject* e *one*, são representados em Alloy por *Comprehension Sets*, onde o iterador *reject* faz uso do operador *not* e o iterador OCL *one* faz uso dos operadores # e =.

O iterador OCL *isUnique* é representado em Alloy por uma fórmula que usa os operadores *all*, *disj* e *!=*. A notação *TR(expr)'* significa que o mapeamento leva em consideração a variável *x'* no lugar da variável *x*.

Finalmente, o iterador OCL *collect* é representado em Alloy pela transformação *TR(expr)*, ao qual substitui a variável *x* da expressão *expr* pela expressão coleção *col*.

OCL	Alloy
col->forAll(x: Type | be) : Boolean	all x: TR(col) | TR(be)
col ->exists(x: Type | be) : Boolean	some x: TR(col) | TR(be)
col->select(x: Type | be) : Set(Type)	{ x: TR(col) | TR(be) }
col->reject(x: Type | be) : Set(Type)	{ x: TR(col) | not TR(be) }
col ->one(x: Type | be) : Boolean	# { x: TR(col) | TR(be) } = 1
col->isUnique(x:Type | expr) : Boolean	all disj x,x': TR(col) | TR(expr) != TR(expr)'
col->collect(x: Type | expr)	TR(expr)

Tabela 7. Mapeamento das expressões de iteradores OCL para Alloy.

4.2.2.3 Restrições

Nesta subseção refinaremos o que foi dito na visão geral do formato de restrições OCL da subseção 4.1

Restrições OCL são definidas por uma expressão OCL aplicada em um contexto. Ela deve ser verdadeira para todas as instâncias do contexto em todos os pontos no tempo, quer sejam invariantes ou derivações. Portanto, restrições OCL sempre especificam restrições a serem satisfeitas em todas as instâncias dos mundos temporais (mundos passados, presentes, contra factuais e futuros).

A Tabela 8 mostra a transformação de cada restrição OCL considerada. A primeira linha da tabela. a transformação de invariantes OCL, a segunda linha a transformação de derivações OCL de finais de associação e por último, derivações OCL de atributos.

OCL	Alloy
context *Classifier* inv invariant-name: *OCLExpression*	all w: World | all self: TR(*Classifier*) | TR(*OCLExpression*)
context *Classifier* :: *Property* : Set(Type) derive: *OCLExpression*	all w: World | all self: TR(*Classifier*) | self.TR(*Property*)[w] = TR(*OCLExpression*)
context *Classifier* :: *Attribute* : Type derive: *OCLExpression*	all w: World | all self: TR(*Classifier*) | self.(w.TR(*Attribute*)) = TR(*OCLExpression*)

Tabela 8. Mapeamento das restrições OCL para Alloy.

A duas primeiras partes dos mapeamentos da invariante e das duas derivações OCL em Alloy , i.e., *all w: World | all self :TR(Classifier) |* , especifica que para todo Mundo (*World*) *w*, a restrição deve ser verdadeira para toda instância do contexto *self* contida no Mundo *w*.

Os mapeamentos de derivações OCL de finais de associação (segunda linha da tabela) e de atributos (terceira linha) se diferenciam devido a diferença entre os seus mapeamentos em Alloy (subseção 4.2.2.1), pois em Alloy atributos são relações ternárias e finais de associaçao são funções. Assim, para finais de associação, a expressão *self.TR(Property)[w] = TR(OCLExpression)* especifica que o final de associação derivado é igual a avaliação da expressão OCL que especifica de onde o valor deve ser derivado. O mesmo vale para os atributos na *expressão self.(w.TR(Attribute)) = TR(OCLExpression*), ou seja, os atributos são iguais a avaliação da expressão OCL

4.3 Validando o Exemplo OntoUML com Restrições de Domínio

Transformando a invariante da subseção 3.2.1 e a derivação da subseção 3.2.2 em Alloy, de acordo com os mapeamentos definidos na subseção anterior, obtemos o seguinte resultado mostrado na Figura 18:

```
1   fact {
2     all w: World | all self: w.RearEndCollision |
3     # self.vehicles[w] = 2
4   }
5   fact {
6     all w: World | all self: w.TravelingVehicle |
7     # { t: self.travel[w].travelers[w] | t in w.Driver } = 1
8   }
9   fact {
10    all w: World | all self: w.RoadTrafficAccident |
11    self.(w.fatalvictims) =
12    # { d: self.victims[w] | d in w.DeceasedPerson }
13  }
```

Figura 18. Transformando a invariante e a derivação OCL do exemplo para Alloy.

A Figura 19 mostra uma possível instanciação para a execução da especificação Alloy do Anexo A (gerada pela transformação OntoUML) adicionada dos fatos Alloy da Figura 18 (as restrições OCL que foram transformadas em fatos).

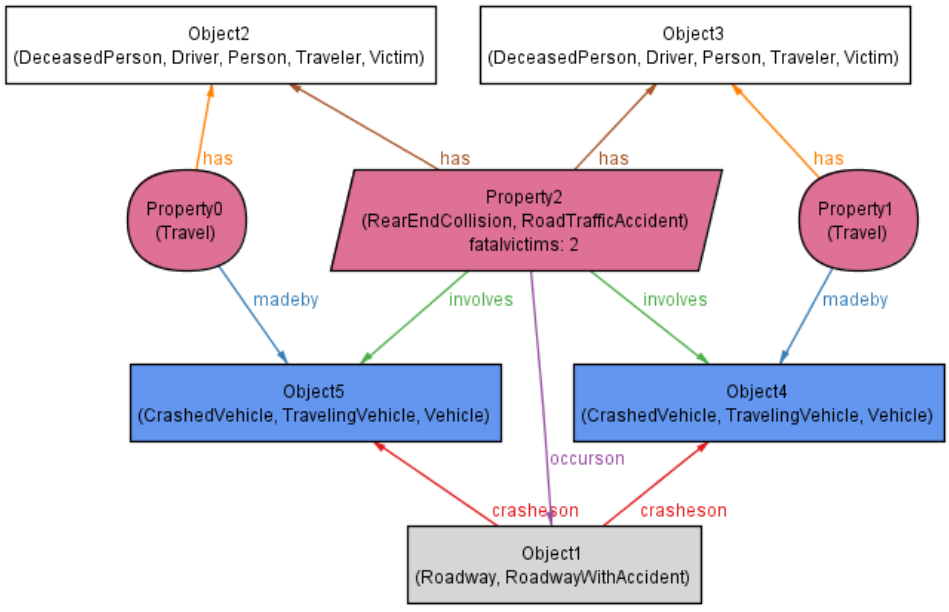

Figura 19. Current World - exemplo OntoUML com restrições.

Esta instanciação considera em um mundo atual (*CurrentWorld*), um acidente de trânsito (*Property2*) do tipo colisão traseira, que ocorre (*occurs on*) em uma estrada (*Object1*), e envolve (*has*) duas vítimas fatais (*Object2* e *Object3*), a saber, ambos motoristas dos veículos acidentados (*Object4* e *Object4*, respectivamente).

Note que agora há uma expressividade maior com relação à conceituação de domínio. Não é permitido que colisões traseiras tenham um número diferente de dois veículos, assim como o número de vítimas fatais agora é sempre derivado do número de vítimas mortas no acidente. Além disso, todo veículo que faz uma viagem sempre possui um motorista.

5 Validação Enriquecida com Restrições OCL

Este capítulo apresenta a estratégia de validação de modelos OntoUML baseada em restrições OCL. A abordagem referida neste capítulo é proveniente de (JACKSON, 2002, p.4, 5). Foi utilizada também em (ANASTASAKIS, 2008) para validar modelos UML com restrições OCL. Nesta abordagem de validação, o modelador pode, não apenas restringir o modelo através de restrições de domínio OCL e visualizar uma possível instanciação gráfica daquele modelo com restrições, mas também pode, usando OCL, simular e checar possibilidades de instanciação do modelo gerando exemplos e contraexemplos do mesmo.

A seção 5.1 descreve a simulação de restrições OCL e a seção 5.2 a asserção de restrições OCL. Por final, a seção 5.3 apresenta a estratégia de validação baseada em restrições OCL.

5.1 Executando Simulações

Se o modelador não pretende transformar as restrições OCL em fatos, ele pode querer analisar o modelo com uma restrição inclusa ou exclusa no modelo (JACKSON, 2012, p.123). Transformando uma restrição OCL em um predicado em Alloy (*predicate*), o modelador é capaz de simular uma instanciação particular a partir do modelo e das outras restrições OCL que já foram transformadas em fatos.

A Figura 20 mostra uma invariante em OCL para simulação em que cada acidente de trânsito (*RoadTrafficAccident*) deve envolver exatamente uma veículo acidentado.

```
context RoadTrafficAccident inv one_vehicle:
self.vehicles->size() =1
```

Figura 20. Invariante OCL para simulação no exemplo OntoUML com restrições.

A Figura 21 mostra a transformação da simulação da invariante OCL para um predicado (*predicate*) em Alloy. A Figura 22 mostra o exemplo encontrado pelo analisador Alloy ao se executar esse predicado.

```
1    pred one_vehicle {
2      all w: World | all self: w.RoadTrafficAccident |
3      # self.vehicles[w] = 1
4    }
5    run one_vehicle for 10 but 3 World, 7 int
```

Figura 21. Transformando a invariante OCL para simulação em Alloy.

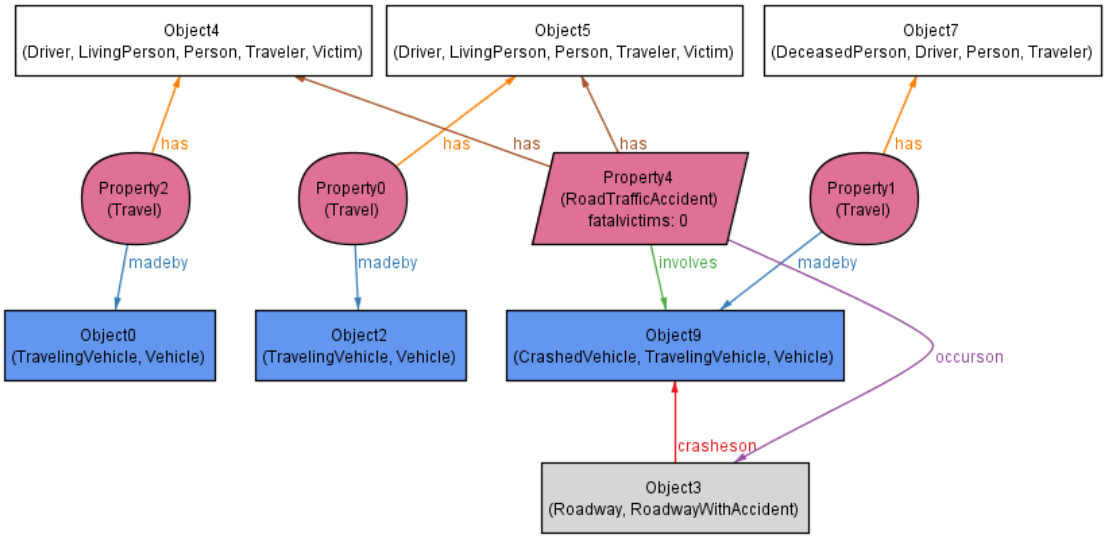

Figura 22. Current World – simulando o exemplo OntoUML com restrições.

Como um exemplo (um cenário) no qual a simulação da invariante OCL é verdadeira, temos um a instanciação que considera em um mundo atual (*CurrentWorld*) um acidente de trânsito (*Property4*), em uma estrada (*Object3*), envolvendo duas vítimas (*Object4*,*Object5*) e um veículo acidentado (*Object9*).

Duas propriedades indesejadas surgem dessa simulação: Primeiro, o viajante (*Object7*) do veículo envolvido no acidente (*Object9*) não é uma vítima neste acidente. Segundo, os veículos *Object0* e *Object2* das vítimas deste acidente não são envolvidos neste acidente.

Através de asserções, como veremos a seguir, podemos confirmar que o modelo (i.e., o exemplo OntoUML adicionado de duas invariantes e uma derivação OCL, como decorrido até aqui) realmente não satisfaz nenhuma dessas propriedades que foram descobertas por essa simulação, ou seja, nem todos os viajantes dos veículos envolvidos em um acidente são vítimas nesse acidente e nem todos veículos das vítimas de um acidente estão envolvidos no acidente.

5.2 Checando Asserções

Asserções são restrições que devem ser satisfeitas a partir dos fatos do modelo (JACKSON, 2012, p.127). Transformando uma restrição OCL em uma asserção Alloy (*assertion*), o modelador é capaz de checar uma propriedade que ele espera que satisfaça a partir do modelo e das restrições OCL que já foram transformadas pra fatos.

A Figura 23 mostra uma invariante em OCL para uma asserção em que os viajantes dos veículos envolvidos em um acidente também são vítimas nesse acidente (vimos pela simulação que essa asserção parece ser falsa).

```
context CrashedVehicle
inv travelers_are_victims_in_accident:
  self.travel.travelers->forAll(t | t.oclIsKindOf(Victim) and
  t.oclAsType(Victim).accident = self.accident)
```

Figura 23. Invariante OCL para asserção do exemplo OntoUML com restrições.

A Figura 24 mostra a transformação da asserção de uma restrição OCL para uma asserção (*assertion*) em Alloy.

```
1  assert travelers_are_victims_in_accident {
2    all w: World | all self: w.CrashedVehicle |
3    all t: self.travel1[w].travelers[w] |
4    t in w.Victim) and (t.accident1[w] = self.accident[w])
5  }
6  check travelers_are_victims_in_accident
7  for 10 but 3 World, 7 int
```

Figura 24. Transformando a Invariante OCL para asserção em Alloy.

A Figura 25 mostra o contraexemplo encontrado pelo analisador Alloy ao se checar essa asserção.

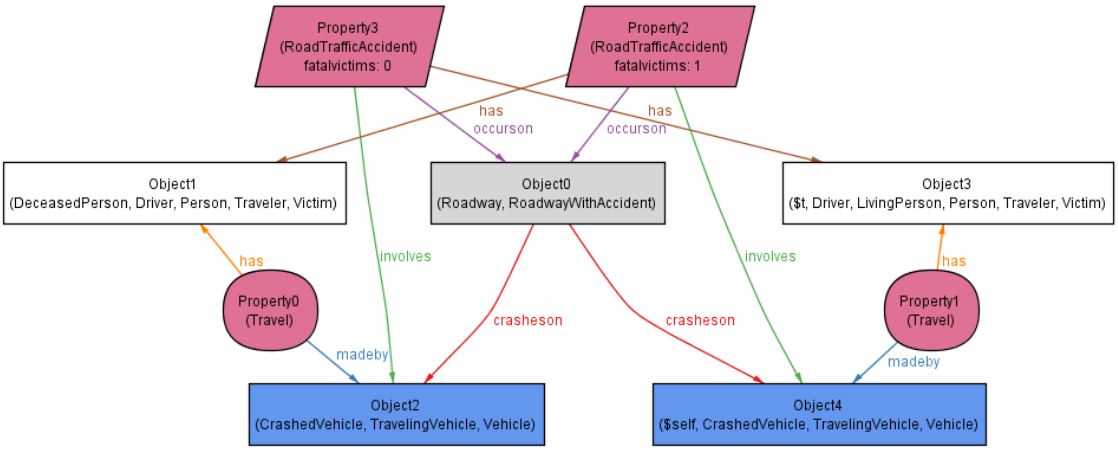

Figura 25. Current World - checando o exemplo OntoUML com restrições.

Como um contraexemplo (um cenário) no qual a asserção da restrição OCL falha, temos uma instanciação que considera um mundo atual (*CurrentWorld*), no qual o viajante e motorista (*Object3*) do veículo (*Object4*) que está envolvido no acidente (*Property2*) não é uma vítima neste acidente. O *label $t* na vítima indica que esta é a instância que não satisfaz a asserção e o *label $self* que esta é a instância do contexto que não satisfaz a asserção.

Como percebido, o modelo (exemplo OntoUML adicionado das duas invariantes e uma derivação OCL) não satisfaz essa propriedade checada. Além disso, esse contraexemplo revela algo ainda mais interessante: esse viajante e motorista (*Object3*) é vítima na verdade de outro acidente (*Property3*), ou seja, uma pessoa é vítima de um acidente X, mas o seu veículo está envolvido em outro acidente Y, em um mesmo ponto no tempo!

5.3 Estratégia de Validação Baseada em Restrições OCL

Simulações OCL podem ajudar a resolver problemas de modelos *overconstrained (*modelos que não permitem estados de objetos considerados admissíveis na concepção do modelador, como ilustrado na parte (c) e (d) da Figura 3). Por exemplo, uma simulação OCL pode ser capaz de reportar que nenhuma instância existe dentro do escopo (*casos*

de testes) mostrando que o modelo está super-restrito ou que a instância existe, porém, em um escopo maior.

Simulações OCL também podem ajudar a resolver problemas de modelos *underconstrained* (modelos que admitem estados de objetos que são inadmissíveis na concepção do modelador, como ilustrado na parte (b) e (d) da Figura 3). Por exemplo, uma simulação OCL pode mostrar instâncias com uma propriedade estrutural indesejada ao modelador.

Semelhantemente, asserções OCL ajudam a resolver problemas de modelos *underconstrained*, i.e., modelos pouco restritos, por exemplo, mostrando um contraexemplo (um cenário) que é aceitável no modelo, mas que viola uma propriedade desejada ao modelador, e modelos *overconstrained*, i.e., modelos super-restritos, mostrando que não existe uma instanciação possível dentro do escopo checado ou que ela existe, porém também em um escopo maior.

Portanto, simulações e asserções OCL viabilizam uma avaliação da acurácia verificando os níveis de precisão e cobertura do modelo conceitual segundo à concepção do modelador. Juntas, elas permitem um processo incremental para produzir modelos. Por exemplo, o modelador pode começar com um modelo conceitual OntoUML mínimo e realizar várias simulações OCL para detectar instanciações indesejadas (não pretendidas pela concepção do modelador) ou para detectar se o modelo está super restrito (nesse caso, o aumento do escopo torna-se necessário). Em seguida, o modelador pode formular se possível, asserções OCL que especifiquem propriedades pretendidas no modelo e realizar a checagem das mesmas à medida que novos fatos OCL são adicionados no modelo conceitual. Contraexemplos resultantes da checagem de asserções OCL sugerem a adição de novos fatos OCL no modelo (JACKSON, 2002, p.4, 5).

Assim, esse processo de validação baseada em restrições OCL, por meio de simulações e checagem de asserções, ajuda a produzir modelos conceituais OntoUML que possuem somente as propriedades desejadas, i.e., a produção de modelos acurados com relação a conceituação de domínio (JACKSON, 2002, p.4, 5).

6 Implementação

Este capítulo descreve os principais aspectos de implementação desse projeto. Mostra a validação com restrições de domínio OCL e a transformações desenvolvidas incorporadas na ferramenta experimental de validação de modelos chamada MOVE (*Model Validation Environment*) https://code.google.com/p/ontouml-lightweight-editor/

A seção 6.1 descreve a abordagem de implementação utilizada. A seção 6.2 descreve o esquema proposto para a transformação da linguagem OCL em Alloy, descrevendo o analisador de restrições OCL, a transformação de OntoUML para UML desenvolvida, e o visitante da sintaxe abstrata das restrições OCL, respectivamente. Finalmente, a seção 6.3 apresenta a incorporação da validação de modelos OntoUML enriquecida com restrições OCL na ferramenta MOVE.

6.1 Abordagem

A transformação das restrições OCL em Alloy foi implementada em Java. Foi usada a implementação EMF de OCL provida pelo projeto MDT (*Model Development Tools*) da Fundação Eclipse (EF) (*Eclipse Foundation*) (http://www.eclipse.org/modeling/mdt/) para analisar e obter as restrições OCL.

Essa implementação de OCL é baseada em modelos EMF, ou seja, baseadas em modelos Ecore ou UML (http://www.eclipse.org/modeling/mdt/downloads/?project=ocl). Isso quer dizer que o metamodelo de OCL possui algumas metaclasses do metamodelo de UML, por exemplo, as metaclasses *Class*, *PrimitiveType*, *Classifier* e etc, o que caracteriza a implementação de OCL/UML do Eclipse (OCL *binding with* UML). O mesmo vale para a implementação OCL/Ecore do Eclipse.

A implementação EMF de OCL/UML (OCL *binding with UML*) provida pelo Eclipse possui um *Visitor (*visitante*)* capaz de inspecionar cada componente da Sintaxe Abstrata (AST-*Abstract Syntax Tree*) de uma restrição OCL viabilizando uma transformação para uma linguagem textual.

Assim a utilização dessas implementações EMF de OCL é dependente de modelos UML (ou Ecore), pois OCL é uma linguagem a ser usada em conjunto com uma notação diagramática, sendo as restrições OCL aplicadas a elementos (classes) dessas linguagens.

A linguagem OntoUML foi implementada em (CARRARETO, 2010) através do framework EMF. Embora proposta como uma extensão *lightweight* da UML em (GUIZZARDI, 2005), OntoUML foi implementada por Carraretto (2010) como uma extensão do metamodelo UML em Eclipse (e não como um perfil UML). Isto foi necessário devido ao suporte inadequado para perfis UML no ambiente Eclipse em 2010. Isto significa que a atual implementação de referência do metamodelo de OntoUML não é um perfil de UML (*UML Profile*) e portanto não pode ser beneficiada da implementação EMF de OCL/UML (*binding with* UML).

Além disso, a implementação de OCL em EMF não pode ser aplicada as instâncias de modelos Ecore . Com isso, os modelos OntoUML que são instâncias do metamodelo da linguagem OntoUML que foi expresso usando a linguagem Ecore não podem ser beneficiados com essa implementação de OCL.

Portanto, como a implementação de OCL baseada em EMF só pode ser usada com a linguagem Ecore ou UML, ela não permite a aplicação de restrições OCL em modelos OntoUML diretamente usando a implementação de (CARRARETO, 2010).

Pode se dizer que, no que diz respeito ao uso de restrições OCL através do framework Eclipse (EMF), a implementação da linguagem OntoUML possui pouca compatibilidade com o ferramental e framework UML do que o esperado quando proposta em (GUIZZARDI, 2005).

A Figura 26 retrata a abordagem utilizada para viabilizar a análise e obtenção das restrições OCL em modelos conceituais OntoUML. Ela utiliza a implementação EMF de

OCL, i.e., a estratégia de transformação do modelo conceitual OntoUML em um modelo UML, a fim de utilizar a implementação OCL *binding with* UML do Eclipse.

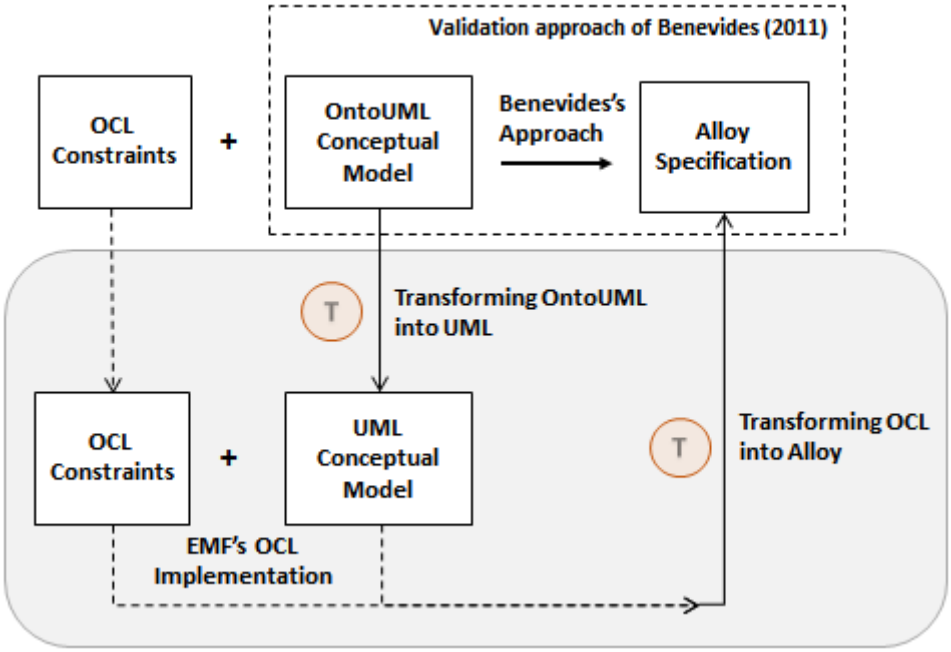

Figura 26. Abordagem proposta para a transformação de OCL para Alloy.

6.2 Esquema

A transformação de restrições OCL é compatível e baseada na transformação do modelo conceitual OntoUML, e portanto, o código Alloy gerado pela transformação OCL deve ser adicionado à especificação Alloy gerada pela transformação OntoUML.

Além disso, a transformação OCL necessita do modelo conceitual OntoUML como entrada (*input*). Por exemplo, um modelo conceitual em OntoUML de *input* seria o modelo de exemplo OntoUML da Figura 6.

As restrições (invariantes e derivações) abordadas durante esse trabalho seriam as restrições OCL a serem transformadas em Alloy e serviriam também como *input* na transformação. As restrições OCL são escritas em formato textual com uma extensão **.ocl*.

A Figura 27 ilustra o esquema de implementação da transformação de OCL em Alloy.

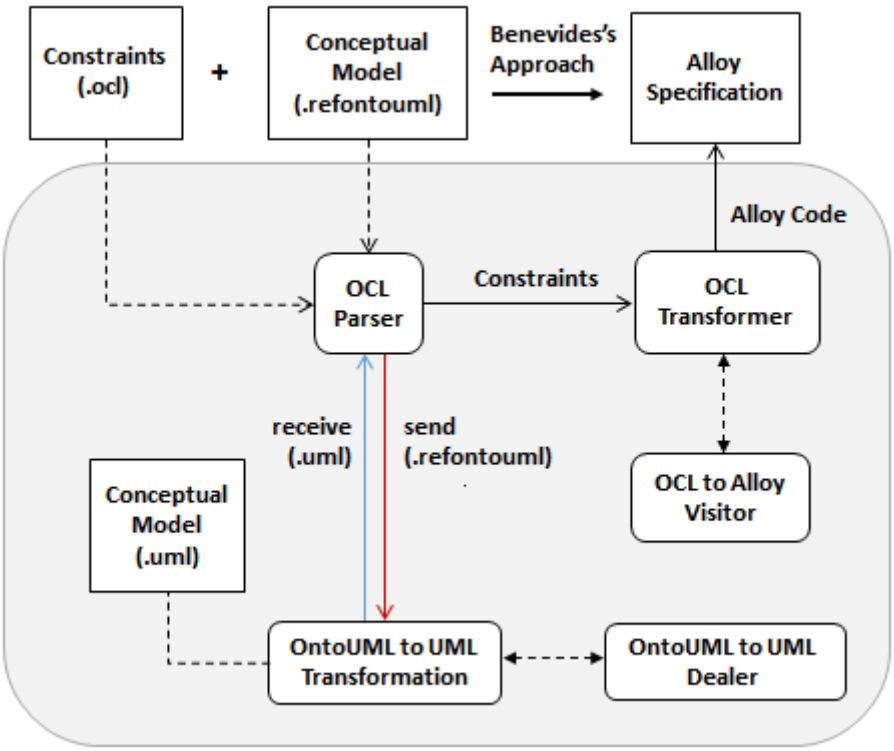

Figura 27. Esquema de implementação da transformação de OCL para Alloy.

6.2.1 Analisador de Restrições OCL

O *Parser* de OCL (analisador de restrições OCL) recebe como entrada o modelo OntoUML e as restrições textuais OCL, e retorna como resultado o conjunto de objetos representado as restrições OCL (*Constraints*). Esses *Constraints* por sua vez são utilizados pelo transformador OCL (*Transformer)* para transformar cada um dos objetos *Constraints* em Alloy, de acordo com as opções de validação, a saber, restrições, simulações ou asserções.

O analisador OCL é responsável por transformar o modelo OntoUML em um modelo UML *backend* e fazer a análise das restrições utilizando o modelo UML gerado através da transformação de OntoUML para UML desenvolvida. Além disso, o *parser* OCL mantém informações dos mapeamentos OntoUML → UML que são úteis para a transformação de OCL para Alloy.

O Anexo B desse projeto apresenta o código Java responsável pela análise de restrições OCL (*OCLParser*), incluindo a chamada da transformação de OntoUML para UML desenvolvida.

6.2.2 A Transformação de OntoUML para UML

A transformação de OntoUML para UML proposta e desenvolvida foi construída tendo como base uma transformação inversa existente de Carraretto (2010).

A Figura 28 mostra o modelo UML *backend* gerado pela transformação do exemplo OntoUML da Figura 6 (mostramos o conteúdo do modelo UML gerado através da plataforma Eclipse, apenas para exemplificar a transformação de OntoUML para UML desenvolvida).

A transformação de Carraretto (2010) era uma transformação no nível de metamodelagem (nível M2), específica de sua infraestrutura, de UML para OntoUML (o processo inverso da transformação referida aqui). Além disso, a transformação de Carraretto (2010) não considerava na transformação, por exemplo, a hierarquia de pacotes (*Packages*) ou Atributos.

A transformação de OntoUML para UML desenvolvida é uma transformação do nível de modelagem (nível M1). A transformação desenvolvida engloba a hierarquia de pacotes (*Packages*), os Atributos de Classes, os Tipos Primitivos (*PrimitiveTypes*), as enumerações (*Enumerations*) e etc. Desconsideramos porém na transformação os comentários UML (*Comments*) por uma questão de simplicidade.

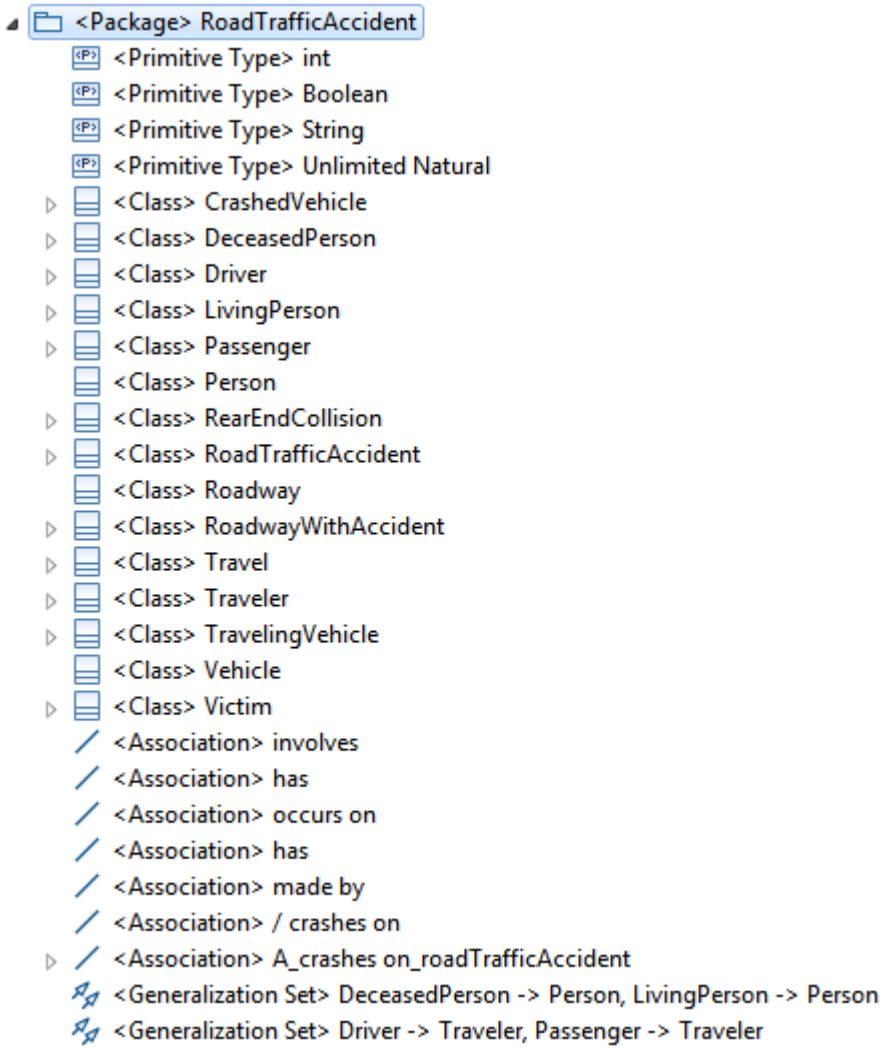

Figura 28. Transformando o exemplo OntoUML em UML.

6.2.3 O Visitante para Alloy

O *Visitor* (visitante) da implementação de OCL/UML (*binding with* UML) é capaz de "percorrer" a sintaxe abstrata de uma restrição OCL e de transformar os elementos de uma restrição OCL em uma linguagem textual (Alloy). Nós estendemos essa implementação do visitante de OCL (a saber, *org.eclipse.ocl.utilities.AbstractVisitor*) com todos os mapeamentos Alloy da seção 4.2.

6.3 Incorporação à Ferramenta de Validação MOVE

A transformação de OCL para Alloy é parte da ferramenta de validação de modelos OntoUML (MOVE) https://code.google.com/p/ontouml-lightweight-editor/, ao qual é uma ferramenta *experimental* para a validação de modelos em OntoUML que é, por enquanto, baseada principalmente na linguagem Alloy como alvo de transformações. A ferramenta também pode ser chamada diretamente do editor OLED (*OntoUML Lightweight Editor*).

A Figura 29 mostra o uso do analisador sintático OCL e apresenta o simples editor embutido na ferramenta para a escrita de restrições OCL em modelos OntoUML.

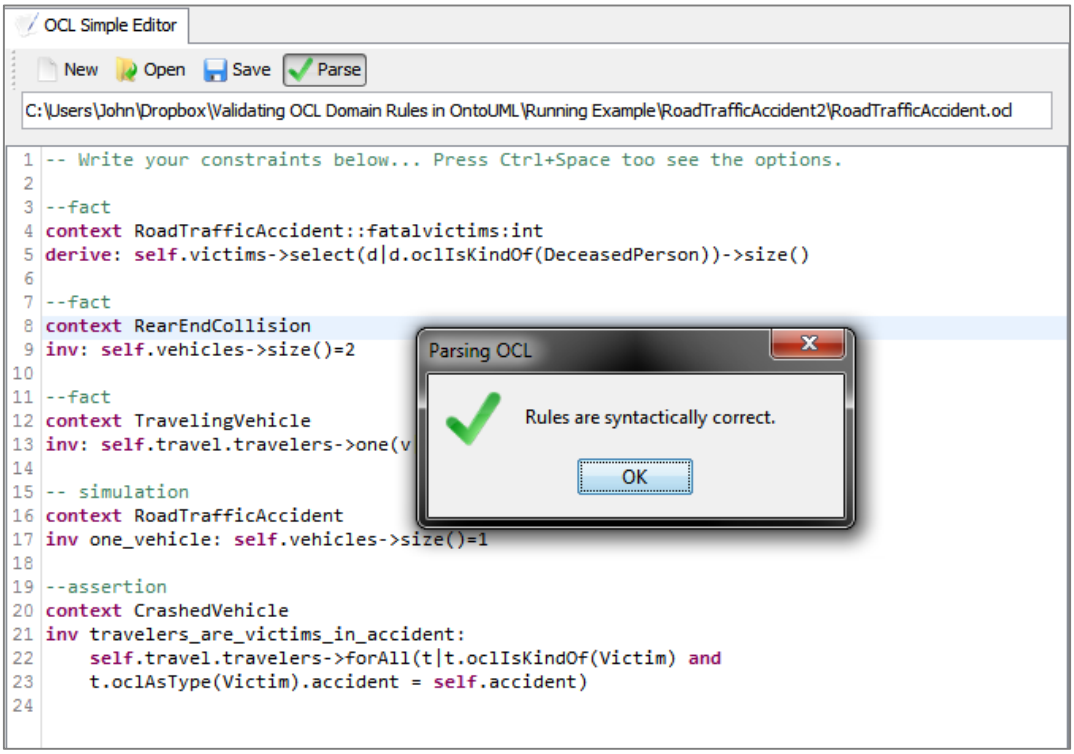

Figura 29. Analisando as restrições OCL na ferramenta MOVE.

A Figura 30 apresenta as opções de validação usando-se restrições OCL, a saber, aplicação de restrições, execução de simulações e checagem de asserções, como apresentado no decorrer deste trabalho.

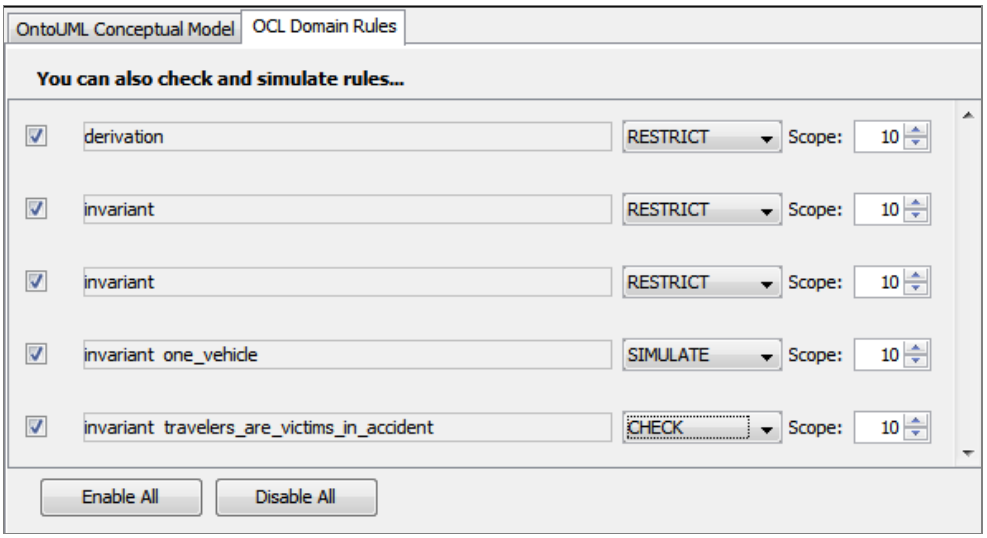

Figura 30. Restrições, simulações e asserções OCL na ferramenta MOVE.

A ferramenta *experimental* MOVE inclui a transformação de OntoUML para Alloy (abordada nesse projeto) e que foi substancialmente melhorada por Sales. Além disso, a ferramenta MOVE também inclui um gerenciador de anti-padrões semânticos desenvolvido por Sales, baseado em Alloy, que detecta, simula e corrige anti-padrões automaticamente através da incorporação de restrições OCL no modelo OntoUML referentes aos padrões encontrados (SALES T.P., 2012).

7 Conclusões

Este capítulo discute as contribuições deste trabalho (seção 7.1), os trabalhos relacionados (seção 7.2) assim como as limitações e possíveis trabalhos futuros (seção 7.3).

7.1 Contribuições

Neste trabalho foi proposto e implementado um mapeamento da linguagem OCL para a linguagem Alloy compatível com OntoUML e uma transformação de restrições OCL para Alloy viabilizando o uso de restrições OCL no processo de validação de modelos OntoUML. Foi discutida uma estratégia de validação utilizando OCL para a validação do modelo OntoUML enriquecida com restrições OCL, e, por fim, foi incorporada a implementação da transformação à ferramenta experimental de validação de modelos MOVE.

Como uma das principais contribuições, este trabalho define um mapeamento entre OCL e Alloy com um maior número de construtos OCL (principalmente iteradores OCL) do que aquele considerado por Anastasakis (2008), mostrando a possibilidade de uma maior interseção entre as duas linguagens.

Além disso, este trabalho contribui para a melhoria da qualidade dos modelos conceituais OntoUML, viabilizando a produção de modelos mais acurados no que diz respeito à representação da conceituação de domínio de acordo com a intenção do modelador. Modelos OntoUML adicionados de restrições de domínio OCL podem agora ser validados através da geração de instâncias fornecidas pela linguagem Alloy e seu analisador.

Além disso, o trabalho contribui para a viabilização de um ambiente de desenvolvimento de restrições OCL em modelos conceituais OntoUML. A escrita de restrições OCL em modelos OntoUML sem um ambiente de desenvolvimento que forneça no mínimo uma verificação sintática da linguagem é desencorajadora. Esse projeto supre essa necessidade ao incorporar na ferramenta de validação MOVE um editor simples de restrições no qual as expressões OCL podem ser verificadas sintaticamente, viabilizando as-

sim a escrita de restrições OCL em modelos OntoUML (através do respectivo modelo UML gerado no *backend*), o que até então não podia ser feito.

A própria transformação de modelos OntoUML em modelos UML, usando a implementação UML do Eclipse (*org.eclipse.uml.uml2*), pode ser vista como uma contribuição indireta desse projeto pois até então não existia uma transformação de modelos OntoUML para UML.

O processo de validação de modelos conceituais OntoUML enriquecidos com restrições é uma tarefa desafiadora uma vez que necessita da previsão de todas as possibilidades de instanciação ao longo da estrutura de mundos do modelo. A validação de modelos OntoUML já é em si uma tarefa difícil devido à complexidade da fundamentação das categorias e axiomas presentes na linguagem, assim, a validação de modelos OntoUML enriquecidos com restrições é ainda mais difícil pois o nível de complexidade aumenta a medida que novas restrições são adicionadas no modelo. Este trabalho também contribui para facilitar esse processo de validação enriquecido com restrições, uma vez que é possível através de OCL executar simulações e checar asserções no modelo. Essa estratégia dá ao modelador mais controle sobre as instâncias e snapshots fornecidos pelo analisador Alloy.

O projeto contribuiu ainda para o refinamento da transformação de OntoUML para Alloy, por exemplo no que diz respeito a transformação de finais de associação (*Association Ends*), a qual não tinha sido incorporada na transformação de modelos OntoUML.

7.2 Trabalhos Relacionados

O principal trabalho relacionado é o de Anastasakis (2008) onde foi definida uma transformação de OCL para Alloy, incorporada com a transformação de UML para Alloy. Nosso trabalho difere do trabalho de Anastasakis principalmente porque a transformação OCL desenvolvida aqui é compatível com OntoUML (e não UML); porque não é incorporada à transformação de modelos OntoUML (não foi desenvolvida como parte dela, como é o caso na transformação de modelos UML de Anastasakis) e sim desen-

volvida separadamente, sendo o seu resultado adicionado ao resultado da transformação OntoUML; e além disso, porque foi definido um conjunto maior de construtos OCL do que aqueles considerados em (ANASTASAKIS et al., 2008, p.74).

A transformação OCL desenvolvida suporta quase todos os iteradores OCL, a saber, os iteradores *forAll*, *exists*, *select*, *reject*, *one*, *isUnique* e *collect*, ao invés de apenas os dois primeiros iteradores. Além disso, com respeito aos tipos especiais, foi suportado os tipos *OclAny* e *OclVoid* e as operações *oclIsTypeOf* e *oclAsType*. Com relação às coleções OCL, foi suportada apenas a coleção do tipo *Set* (Anastasakis suporta apenas coleções do tipo *Set* também). Com relação a Inteiros e Booleanos, o suporte é semelhante ao de Anastasakis, sendo que, com relação às operações, nossa transformação OCL suporta quase todas as operações de tipos Booleanos e Inteiros, ao contrário de Anastasakis (ver seção 4.2).

7.3 Limitações e Trabalhos Futuros

O mapeamento de um valor Booleano para Alloy não é simples; um literal booleano *true* em OCL pode ser transformado para valores booleanos diferentes em Alloy dependendo do contexto no qual está inserido. Nesse caso, adotamos uma abordagem simplista não suportando literais booleanos (*true* ou *false*) ou atributos booleanos do modelo (ver subseção 4.2.1.3). Uma abordagem mais sofisticada poderia ser investigada para garantir uma cobertura abrangente de expressões envolvendo booleanos.

Além disso, o iterador OCL *closure* não foi considerado nos mapeamentos, pois esse iterador foi recentemente incorporado nas implementações EMF de OCL do Eclipse. O iterador OCL *closure* é aplicado em uma coleção e resulta no conjunto de descendentes da coleção de acordo com um relacionamento. Por exemplo, a seguinte expressão OCL *"parents->closure(children)"* resulta na união dos conjuntos *"parents.children"*, *"parentes.children.children"*, e assim por diante. Esse iterador é importante na modelagem conceitual OntoUML (na modelagem de linguagens derivadas de UML), devido aos possíveis *self-relationships* (auto relacionamentos) do modelo que podem adquirir um comportamento cíclico (relacionamentos que possuem como domínio e contra do-

mínio uma mesma classe, por exemplo, um pai que é descendente dele mesmo). Um mapeamento direto para esse iterador OCL se daria através do operador *transitive closure* de Alloy (sintaxe concreta: ^). Por exemplo, o resultado em Alloy do mapeamento da expressão OCL supracitada seria algo como: *TR(parentes).^TR(children)*.

As restrições OCL são capazes de especificar regras que devem ser satisfeitas em todos os pontos no tempo (seção 4.1) (subseção 4.2.2.3). OCL não é capaz de especificar restrições de domínio temporais, ou seja, regras a serem satisfeitas envolvendo diferentes pontos no tempo (diferentes Mundos). É fácil pensar em uma conceituação em que essas possiblidades de instanciação são indesejáveis, por exemplo, um adulto que passa a ser criança, i.e., uma pessoa que desempenha a fase de adulto em um Mundo Presente w, mas desempenha a fase de criança num Mundo Futuro *w'* acessível a partir de w, ou uma pessoa morta que volta a viver, i.e., uma pessoa morta em um Mundo Passado (*World*) *w* que torna a viver em um Mundo Futuro *w'* acessível a partir de *w*.

Portanto, o modelo necessitaria de uma restrição que especificasse uma regra envolvendo possibilidades entre Mundos Temporais. Não só em restrições que dizem respeito a fases (*phases*), mais que isso, em regras envolvendo possibilidades de situações entre diferentes pontos no tempo. Ainda é necessário pesquisas e estudos futuros para a especificação deste tipo de regra, visando uma transformação para Alloy e viabilizando a validação desses aspectos da conceituação.

Uma área de atual estudo é a visualização das instâncias geradas pelo analisador Alloy, pois ficamos restritos ao formato de visualização fornecido pelo analisador e a linguagem Alloy. Uma primeira iniciativa nesse sentido é o trabalho de Braga (2011) ao qual investiga o design de diagramas, em particular, os diagramas de instâncias e analisa seu papel na teoria da percepção e processamento cognitivo de imagens, com o propósito de tornar os diagramas de instâncias eficientes do ponto de vista cognitivo. Alloy possui formas de se personalizar essas visualizações, porém ainda é bastante ineficiente cognitivamente e limitada. Além disso, quando cresce o número de instâncias, aumenta-se a dificuldade de avaliar os snapshots fornecidos pelo analisador, isto, devido a vários fatores como disposição, tamanho, cores, formatos, e etc. Assim, planejamos fornecer

uma experiência melhor ao modelador (ou *stakeholder*) através de melhorias na apresentação gráfica dos *snapshots*.

Outra área atual de estudo é a fundamentação ontológica de Tipos de Dados (*DataTypes*) na linguagem OntoUML. A pesquisa é desenvolvida por Antognoni Albuquerque e possui impactos diretos com o desenvolvimento desse projeto uma vez que parte da linguagem OCL é composta de tipos de dados, a saber, tipos primitivos, como booleanos, inteiros ou tipos de dados compostos como *DataTypes*, ou até enumerações, e que hoje fazem parte de OntoUML pois são provenientes da linguagem UML.

Bibliografia

ANASTASAKIS K., BORDBAR B., GEORG G. RAY I. (2008) On Challenges of Model Transformation from UML to Alloy. Available from Internet: http://www.cs.bham.ac.uk/~bxb/Papres/sosym08.pdf

ANDONI A., DANILIUC D., KHURSHID S., MARINOV D. (2002) Evaluating the Small Scope Hypothesis. Available from Internet: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.4000&rep=rep1&type=pdf

BARCELOS, P.P.F., GUIZZARDI, G., GARCIA, A.S., MONTEIRO, M. (2011) Ontological Evaluation of the ITU-T Recommendation G.805. 18th International Conference on Telecommunications (ICT 2011). pp. 266–271. IEEE Press, Ayia Napa, Cyprus.

BENEVIDES A. B., GUIZZARDI G., BERNARDO B. F. B., ALMEIDA J.P., (2011) Validating modal aspects of OntoUML conceptual models using automatically generated visual world structures. Available from Internet: http://www.jucs.org/doi?doi=10.3217/jucs-016-20-2904

BRAGA B. F. B. (2011) Cognitive Effective Instance Diagram Design. Trabalho de Conclusão de Curso. (Graduação em Ciência da Computação) - Universidade Federal do Espírito Santo.

BRAGA, B. F. B.; ALMEIDA, J. P. A.; GUIZZARDI, G.; BENEVIDES, A. B. (2010) Transforming OntoUML into Alloy: Towards conceptual model validation using a lightweight formal method. Innovations in Systems and Software Engineering (ISSE), Springer-Verlag, London, vol. 6, no. 1, p. 55–63, 2010. (Print). Available from Internet: http://www.springerlink.com/content/m1715n1220717l58/fulltext.pdf

CABOT J. and GOGOLLA M. (2012). Object constraint language (OCL): a definitive guide. In Proceedings of the 12th international conference on Formal Methods for the Design of Computer, Communication, and Software Systems: formal methods for model-driven engineering (SFM'12), Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio (Eds.). Springer-Verlag, Berlin, Heidelberg, 58-90. DOI=10.1007/978-3-642-30982-3_3 http://dx.doi.org/10.1007/978-3-642-30982-3_3

CARRARETTO R. (2010) A Modeling Infrastructure for OntoUML. http://rcarraretto.googlecode.com/files/aModelingInfrastructureForOntoUML.pdf

GONÇALVES, B., GUIZZARDI, G., PEREIRA FILHO, J.G. (2011) Using an ECG reference ontology for semantic interoperability of ECG data. Journal of Biomedical Informatics. 44, 126–136.

GUARINO N. (2004) Toward Formal Evaluation of Ontology Quality. IEEE Intelligence Systems 19(4):78-79.

GUIZZARDI, G. (2007) On Ontology, ontologies, Conceptualizations, Modeling Languages, and (Meta) Models. Frontiers in Artificial Intelligence and Applications, Databases and Information Systems IV. Amsterdam: IOS Press.

GUIZZARDI, G. (2005) Ontological foundations for structural conceptual models. Thesis (Ph.D.) - University of Twente, Enschede, The Netherlands, Enschede, October 2005. Available from Internet: http://doc.utwente.nl/50826/ . Cited 01/13/2009

GUIZZARDI, G., BAIÃO, F.A., LOPES, M., DE ALMEIDA FALBO, R. (2010) The Role of Foundational Ontologies for Domain Ontology Engineering: An Industrial Case Study in the Domain of Oil and Gas Exploration and Production. International Journal of Information Systems Modeling and Design (IJISMD). 1, 1–22.

JACKSON, D. (2012) Software abstractions: logic, language, and analysis - Revised Edition. [S.l.]: MIT Press. ISBN 978-0-262-01715-2.

JACKSON, D. (2002) Alloy: a lightweight object modeling notation. Transactions on Software Engineering and Methodology (TOSEM), ACM, New York, NY, USA, vol. 11, no. 2, p. 256–290. ISSN 1049-331X.

MYLOPOULOS, J. (1992) Conceptual modeling, databases, and case: An integrated view of information systems development. In . Chichester: John Wiley & Sons. cap. Conceptual Modeling and Telos, p. 49–68.

OMG, OCL Version 2.3.1 (2012). Document id: formal/2012-01-01 Available from: http://www.omg.org/

OMG UML Version 2.0 (2005). Document id: formal/2005-07-05 Available from: http://www.omg.org/

SALES, T.P., BARCELOS, P.P.F., GUIZZARDI, G. (2012) Identification of Semantic Anti-Patterns in Ontology-Driven Conceptual Modeling via Visual Simulation. 4th International Workshop on Ontology-Driven Information Systems (ODISE), together with the 7th International Conference on Formal Ontology in Information Systems (FOIS). , Graz, Austria.

UML2ALLOY Website (2009) http://www.cs.bham.ac.uk/~bxb/UML2Alloy

WARMER, J. B.; KLEPPE, A. G. (2003) The Object Constraint Language: Getting Your Models Ready for MDA. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. 240 p. ISBN 0321179366.

Anexo A

[Exemplo OntoUML em Alloy]

```
module RoadTrafficAccident

open world_structure[World]
open ontological_properties[World]
open util/relation
open util/boolean
open util/ternary

sig Object {}
sig Property {}

abstract sig World {
  exists: some Object+Property,
  RearEndCollision: set exists:>Property,
  CrashedVehicle: set exists:>Object,
  Passenger: set exists:>Object,
  DeceasedPerson: set exists:>Object,
  Vehicle: set exists:>Object,
  TravelingVehicle: set exists:>Object,
  Victim: set exists:>Object,
  Driver: set exists:>Object,
  Roadway: set exists:>Object,
  Person: set exists:>Object,
  Traveler: set exists:>Object,
  RoadTrafficAccident: set exists:>Property,
  Travel: set exists:>Property,
  RoadwayWithAccident: set exists:>Object,
  LivingPerson: set exists:>Object,
  crasheson: set RoadwayWithAccident one -> some CrashedVehicle,
  has: set RoadTrafficAccident one -> some Victim,
  occurson: set RoadTrafficAccident some -> one RoadwayWithAccident,
  madeby: set Travel one -> one TravelingVehicle,
  has1: set Travel one -> some Traveler,
  involves: set RoadTrafficAccident one -> some CrashedVehicle,
  fatalvictims: set RoadTrafficAccident set -> one Int
}{
  disj[Person,Vehicle+Roadway]
  disj [Vehicle,Person+Roadway]
  disj [Roadway,Person+Vehicle]
  disj [RoadTrafficAccident,Travel]
  disj [Travel,RoadTrafficAccident]
  exists:>Object in Vehicle+Roadway+Person
  disj [Vehicle,Roadway,Person]
  exists:>Property in RearEndCollision+Travel+RoadTrafficAccident
}
```

```
fact additionalFacts {
  linear_existence[exists]
  all_elements_exists[Object+Property,exists]
}

fact rigidity {
  rigidity[Person,Object,exists]
}
fact rigidity {
  rigidity[RearEndCollision,Property,exists]
}
fact rigidity {
  rigidity[Vehicle,Object,exists]
}
fact rigidity {
  rigidity[RoadTrafficAccident,Property,exists]
}
fact rigidity {
  rigidity[Travel,Property,exists]
}
fact rigidity {
  rigidity[Roadway,Object,exists]
}
fact generalization {
  RoadwayWithAccident in Roadway
}
fact generalization {
  CrashedVehicle in TravelingVehicle
}
fact generalization {
  DeceasedPerson in Person
}
fact generalization {
  TravelingVehicle in Vehicle
}
fact generalization {
  RearEndCollision in RoadTrafficAccident
}
fact generalization {
  Traveler in Person
}
fact generalization {
  Passenger in Traveler
}
fact generalization {
  Victim in Traveler
}
fact generalization {
  Driver in Traveler
}
```

```
fact generalization {
  LivingPerson in Person
}
fact generalizationSet {
  Traveler = Passenger+Driver
  disj[Driver,Passenger]
}
fact generalizationSet {
  Person = DeceasedPerson+LivingPerson
  disj [DeceasedPerson,LivingPerson]
}

fun visible : World some -> some univ {
  exists+select13[fatalvictims]
}

fact derivations {
  derivation[crasheson,involves,occurson]
}

fun accident1 [x: World.Victim,w: World] : set
World.RoadTrafficAccident {
  (w.has).x
}

fun victims [x: World.RoadTrafficAccident,w: World] : set World.Victim
{
  x.(w.has)
}

fun travel1 [x: World.TravelingVehicle,w: World] : set World.Travel {
  (w.madeby).x
}

fun vehicle [x: World.Travel,w: World] : set World.TravelingVehicle {
  x.(w.madeby)
}

fun travel [x: World.Traveler,w: World] : set World.Travel {
  (w.has1).x
}

fun travelers [x: World.Travel,w: World] : set World.Traveler {
  x.(w.has1)
}

fun accident [x: World.CrashedVehicle,w: World] : set
World.RoadTrafficAccident {
  (w.involves).x
}
```

```
fun vehicles [x: World.RoadTrafficAccident,w: World] : set
World.CrashedVehicle {
  x.(w.involves)
}

fact relatorConstraint {
  all w: World | all x: w.RearEndCollision | #
(x.(w.involves)+x.(w.has)+x.(w.occurson)) >= 2
}

fact relatorConstraint {
  all w: World | all x: w.RoadTrafficAccident | #
(x.(w.involves)+x.(w.has)+x.(w.occurson)) >= 2
}

fact relatorConstraint {
  all w: World | all x: w.Travel | # (x.(w.has1)+x.(w.madeby)) >= 2
}

fact associationProperties {
  immutable_target[RoadTrafficAccident,has]
  immutable_target[RoadTrafficAccident,occurson]
  immutable_target[Travel,madeby]
  immutable_target[Travel,has1]
  immutable_target[RoadTrafficAccident,involves]
}

run { } for 10 but 3 World, 7 int
```

Anexo B

[Analisador OCL]

```
// OCL Parser Constructor
public OCLParser (String oclAbsolutePath, String refAbsolutePath)

throws IOException,ParserException,Exception
{

  OntoUMLParser refparser = new OntoUMLParser(refAbsolutePath);

  umlResource = OntoUML2UML.Transformation(
      refparser,refAbsolutePath.replace(".refontouml" , ".uml"),true
  );
  umlHashMap = OntoUML2UML.transformer.mydealer.mymap;
  logDetails = OntoUML2UML.logDetails;

  // this line was added due to a bug of Eclipse :
  // https://bugs.eclipse.org/bugs/show_bug.cgi?id=372258
  Environment.Registry.INSTANCE.registerEnvironment(
      new UMLEnvironmentFactory().createEnvironment()
  );

  org.eclipse.uml2.uml.Package umlmodel = (org.eclipse.uml2.uml.Package)
  umlResource.getContents().get(0);
  umlResource.getResourceSet().getPackageRegistry().put(null,umlmodel);

  org.eclipse.ocl.uml.OCL.initialize(umlResource.getResourceSet());

  org.eclipse.ocl.uml.UMLEnvironmentFactory envFactory =
      new org.eclipse.ocl.uml.UMLEnvironmentFactory(
          umlResource.getResourceSet()
      );
  umlenv = envFactory.createEnvironment();
  org.eclipse.ocl.uml.OCL myOCL= org.eclipse.ocl.uml.OCL.newInstance(umlenv);
  myOCL.setParseTracingEnabled(true);

  analyzer = myOCL.createAnalyzer(FileUtil.readFile(oclAbsolutePath));
  cstree = analyzer.parseConcreteSyntax();

  if (cstree!=null && cstree.getStartToken().toString().equals("context"))
  {
      oclConstraints = "package "+umlmodel.getName()+"\n\n" +oclConstraints+
      "\n endpackage\n\n";
  }

  InputStream input = new FileInputStream(oclAbsolutePath);
  org.eclipse.ocl.OCLInput document = new org.eclipse.ocl.OCLInput(input);

  umlconstraintsList = myOCL.parse(document);
  umlreflection = umlenv.getUMLReflection();
}
```