Universidade Federal do Rio de Janeiro

Bruno de Moura Araujo

Um método para Validar a Conformidade de Processos de Negócio com Regras de Negócio

DISSERTAÇÃO DE MESTRADO

Instituto de Matemática

Núcleo de Computação Eletrônica

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

BRUNO DE MOURA ARAUJO

Um método para Validar a Conformidade de Processos de Negócio com Regras de Negócio

Prof. Eber Assis Schmitz, Ph.D.
PPGI/DCC/IM - UFRJ
Orientador

Prof. Alexandre Luis Correa, D.Sc.
CCET - UNIRIO
Co-orientador

Rio de Janeiro, junho de 2010

Ficha Catalográfica

Araujo, Bruno de Moura

Um método para Validar a Conformidade de Processos de Negócio com Regras de Negócio / Bruno de Moura Araujo. – Rio de Janeiro: PPGI/DCC/IM UFRJ, 2010.

122 f.: il.

Dissertação de Mestrado (Mestrado em Informática) – Universidade Federal do Rio de Janeiro. Programa de Pós-Graduação em Informática, Rio de Janeiro, BR–RJ, 2010.

Orientador: Eber Assis Schmitz, Ph.D. PPGI/DCC/IM - UFRJ; Co-orientador: Alexandre Luis Correa, D.Sc. CCET - UNIRIO.

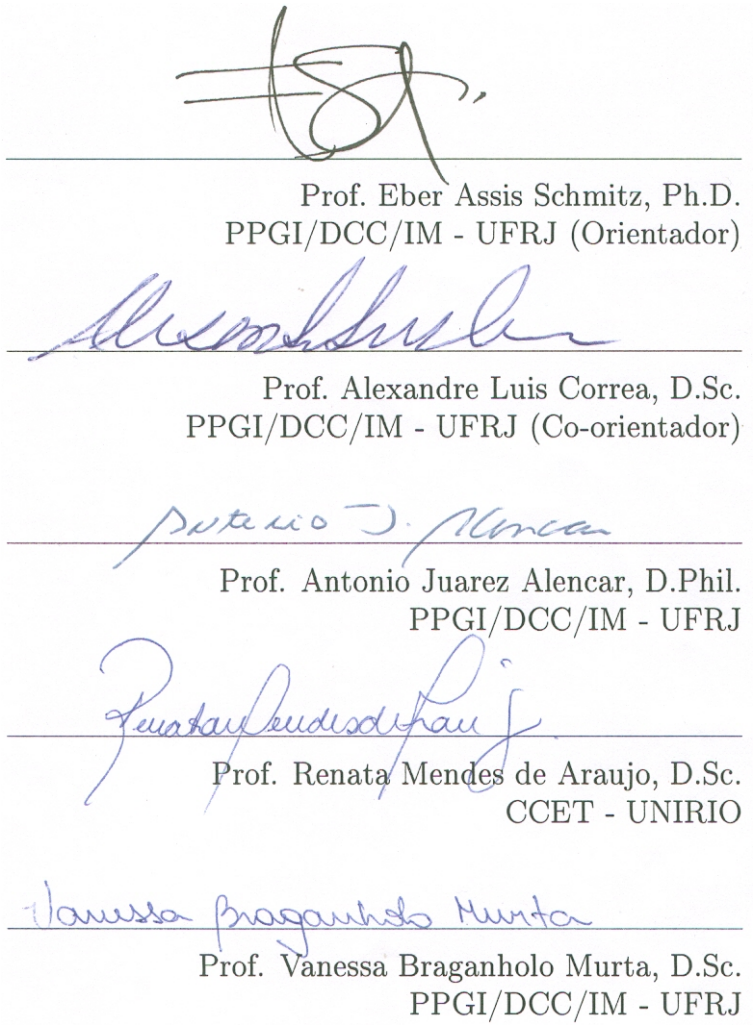
I. Schmitz, Ph.D., Eber Assis. II. Correa, D.Sc., Alexandre Luis. III. Título.

Um método para Validar a Conformidade de Processos de Negócio com Regras de Negócio

Bruno de Moura Araujo

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Informática do Departamento de Ciência da Computação, Instituto de Matemática e Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Aprovado por:



Prof. Eber Assis Schmitz, Ph.D.
PPGI/DCC/IM - UFRJ (Orientador)

Prof. Alexandre Luis Correa, D.Sc.
PPGI/DCC/IM - UFRJ (Co-orientador)

Prof. Antonio Juarez Alencar, D.Phil.
PPGI/DCC/IM - UFRJ

Prof. Renata Mendes de Araujo, D.Sc.
CCET - UNIRIO

Prof. Vanessa Braganholo Murta, D.Sc.
PPGI/DCC/IM - UFRJ

Rio de Janeiro, junho de 2010

A Deus, minha família e amigos.

AGRADECIMENTOS

Agradeço a **DEUS**, sem o qual não teria a mínima condição e capacidade de realizar este trabalho, e a quem devo tudo o que tenho e sou.

À minha **família**, principalmente aos meus pais **Adilson** e **Solange** que, mesmo estando distante, sempre se preocuparam comigo e me dera total apoio sempre que necessário, às vezes abrindo mão de suas próprias necessidades para me ajudar nas minhas.

À **CAPES**, pelo apoio financeiro cedido através da bolsa de estudos.

Aos **professores**, pelo conhecimento que adquiri cursando as disciplinas.

Aos meus orientadores **Eber** e **Alexandre**, por todas as instruções e acompanhamento a este trabalho.

Às secretárias tia **Deise** e **Zezé**, pela dedicação em seu trabalho na secretaria e pelos inúmeros atendimentos que me prestaram com carinho e eficiência desde a graduação.

Aos meus colegas de turma que compartilharam de bons e maus momentos, em especial **Eliêmia**, **Patrícia**, **Leandro** e **Guga**, que me acompanham de perto desde a graduação.

Novamente à **Patrícia** e **Eliêmia**, por todas as idas à secretaria para solicitar documentos por mim, pelas impressões de trabalhos, pela busca dos professores para entregar os trabalhos em mãos, pela companhia no NCE...

À minha namorada, **Sheila**, por sua compreensão e apoio durante essa jornada.

À **EMGEPRON**, por permitir minha e apoiar minha viagem ao SAC 2010 e pelos horários de expediente em que me permitiu estar na UFRJ.

Aos professores **Juarez**, **Renata** e **Vanessa**, por aceitarem o convite de participar da banca de defesa deste trabalho.

RESUMO

A conformidade regulamentar de processos e regras de negócio têm se tornado uma área de grande preocupação para gerentes pois, apesar de ser uma tarefa difícil e dispendiosa, alcançar tal conformidade pode ser uma fonte de diferenciação capaz de adicionar valor considerável ao negócio. Muitas vezes os processos de negócio acabam violando regras do negócio. Essa violações normalmente não são percebidas assim que ocorrem, e podem se acumular, comprometendo assim a conformidade entre as regras e os processos de negócio.

Este trabalho apresenta um método para validar processos de negócio em relação às regras de negócio. No método proposto, processos de negócio são modelados com diagramas de atividades da UML, enquanto regras de negócio são representadas como pré e pós condições em OCL anexadas a atividades do processo ou como invariantes OCL associadas a um modelo de negócio conceitual. A validação do modelo é baseada na simulação de um conjunto de cenários. Cada cenário define uma configuração inicial das instâncias do modelo conceitual necessária para uma análise particular. As ações associadas a uma atividade definida no processo são executadas com o auxílio da ferramenta PRUV, desenvolvida com base na ferramenta USE. Depois da execução de cada atividade as regras de negócio associadas são validadas e qualquer violação existente é detectada. O método proposto permite ao usuário ter uma resposta rápida de possíveis falhas que possam existir em um modelo de processo.

Palavras-chave: MDA, UML, OCL, Processo de Negócio, Regras de Negócio, Validação de Conformidade, Simulação.

A method for Validating the Compliance of Business Processes to Business Rules

ABSTRACT

The compliance between business processes and business rules has become an area of great concern for the managers. Although the task of achieve such compliance is hard and expensive, it can be a source of differentiation capable of adding considerable value to the business. Many times the business processes violate the business rules. Such violations are not normally perceived in the moment they occur and they accumulate, undermining the compliance between the business rules and business processes.

This work presents a method for validating business processes with respect to the business rules. In the proposed method, business processes are modeled with UML activity diagrams, whilst business rules are represented as OCL pre and post conditions attached to process activities or as OCL invariants associated to a business conceptual model. The model validation is based on the animation of a set of scenarios. Each scenario defines an initial configuration of the instances of the conceptual model required for a particular analysis. The actions associated to an activity defined in the process are performed with the help of the tool PRUV, developed based on the tool USE. After the execution of each activity, the associated business rules are evaluated and any existing violations are detected. The proposed method allows the modeler to have an early feedback of possible defects that may exist in a business process model.

Keywords: UML, OCL, Business Process, Business Rules, Compliance Validation, Simulation.

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ASSL	A Snapshot Sequence Language
CG	Condição de Guarda
CSP	Constraint Satisfaction Problem
DA	Diagrama de Atividades
EPC	Event-driven Process Chains
MDA	Model Driven Architecture
MOF	Metamodel Object Facility
MPBF	Modelo de Processo Bem Formado
OCL	Object Constraint Language
OMG	Object Management Group
RN	Regras de Negócio
SBVR	Semantics Business Vocabulary and Rules
SQL	Structured Query Language
USE	UML-Based Specification Environment
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização

Nos últimos anos, a importância da conformidade entre as operações e as normas das organizações tem crescido drasticamente em todas as áreas de negócio (SADIQ; GOVERNATORI; NAMIRI, 2007). Não só a conformidade com a legislação tem sido exigida em vários países, mas também órgãos reguladores têm aumentado suas exigências, demandando que certos tipos de organizações operem de acordo com normas específicas. *Basel Accords* (DECAMPS; ROCHET; ROGER, 2004), *Sarbanes-Oxlay Act* (ROMANO, 2005) e *HIPAA*[1] (ANNAS, 2003) são exemplos conhecidos de tais normas e legislações.

Além disso, em tempos turbulentos como os que o mercado financeiro está experimentando em todo o mundo, investidores apreensivos e acionistas querem ser constantemente tranquilizados de que as organizações nas quais eles têm participação estão operando de acordo com os conjuntos de normas e regras (MADDEN, 2007). Sem mencionar as associações de consumidores que, conscientes de seu poder para influenciar a decisão de consumidores e legisladores, são usadas para reclamar ener-

[1] Health Insurance Portability and Accountability Act

gicamente por garantias de que certas práticas profissionais estão sendo adotadas pelas organizações que vendem os produtos e serviços no qual estão interessados (ARMSTRONG, 2006).

Assim, a conformidade regulamentar de processos de negócio e práticas têm se tornado uma área de grande preocupação para gerentes (LIU; MÜLLER; XU, 2007). Apesar de geralmente se aceitar que integrar conformidade dentro da estrutura de uma organização é uma tarefa difícil e dispendiosa, integração de conformidade pode ser a fonte de diferenciação capaz de adicionar valor considerável ao negócio, agindo como um catalisador para atrair novos recursos, vender mais produtos, reforçar a imagem e a reputação, e atrair clientes (SCHNEIDER, 2007).

O dinamismo dos ambientes de trabalho atuais está impulsionando muitas organizações a moverem-se da estrutura clássica departamental hierárquica para uma estrutura de processo de negócio mais flexível (HALVEY; MELBY, 2007). Como resultado, conformidade de negócio se torna uma consequência direta de conformidade dos processos de negócio.

1.2 Motivação

Um dos motivos que impede que a conformidade de negócio seja alcançada é que muitas vezes os processos de negócio acabam violando regras do negócio. Essa violações normalmente não são percebidas assim que ocorrem, mas apenas quando o processo se encerra de forma inesperada ou quando verificações periódicas são executadas. Essas violações não detectadas imediatamente podem se acumular, tornando-se impossível de serem detectadas e corrigidas de forma precisa posteriormente.

Além disso, em um ambiente tão competitivo, novos processos têm de ser constan-

temente criados e os já existentes repetidamente revisados (LIU; MÜLLER; XU, 2007). Com isso, um crescente número de organizações está automatizando seus processos de negócio para fazer face à concorrência, tornarem-se mais ágeis, ganhar eficiência e reduzir custos (KIRCHMER; SCHEER, 2004). Tudo isso necessita de métodos automatizados para verificação de conformidade de processos (MUEHLEN; INDULSKA; KAMP, 2007).

Porém, os métodos existentes atualmente para se realizar verificações de conformidade como, por exemplo, a técnica de diagrama de estados (MUTH et al., 1998), modelos baseados em rede de Petri (AALST, 2003), e lógica temporal (ESHUIS; WIERINGA, 2001) utilizam especificações formais, que, se por uma lado permitem um estudo detalhado do comportamento de processos, por outro lado são muito difíceis para que pessoas sem formação técnica entendam, pois muitos desses métodos baseiam-se em rigorosos formalismos matemáticos (MUEHLEN; INDULSKA; KAMP, 2007).

1.3 Objetivo

O objetivo deste trabalho é apresentar um método que seja capaz de validar a conformidade de processos de negócio com regras de negócio que seja de mais fácil entendimento que técnicas já existentes, que exigem um profundo conhecimento teórico e de métodos formais para que sejam utilizadas. Ao mesmo tempo, buscamos realizar a validação com maior eficácia na identificação e correção dos erros, já que no método situações reais são simuladas. Assim o erro é detectado no momento em que ocorre e com isso ele pode ser facilmente analisado, entendido e corrigido.

O método utiliza diagrama de classes e de atividades da UML, regras em OCL e uma linguagem de ações para executar automaticamente casos de testes que simulam a

execução real do processo. Isto permite que o usuário seja capaz de identificar e corrigir os erros nos modelos que levam às violações de regras.

1.4 Trabalhos Relacionados

Como trabalhos relacionados, podemos citar, principalmente, os trabalhos que buscam validar processos de negócio através de abordagens formais. Dentre estes existe a abordagem de lógica temporal, adotada por Eshuis e Wieringa (2001). Neste trabalho, os autores baseiam-se na semântica de grafos de estados extendidos com propriedades transacionais para lidar com manipulação de objetos. Já Yang e Zhang (2003) utilizam π-calculus para descrever interações entre processos e representar modelos de fluxos de trabalho a fim de verificar propriedades do modelo. Schroeder (1999) traduz processos de negócio para a álgebra de processos CCS (*Calculus of Communicating Systems*). Existem várias abordagens utilizando redes de Petri, entre as quais podemos citar o trabalho de Van der Aalst (2003). Outras abordagens utilizam *State Chats* ou *Activity Chats* (MUTH et al., 1998). Todos estes trabalhos apresentam um grau de dificuldade alto para serem usados, já que utilizam forte fundamento matemático, que os projetistas de negócio normalmente não possuem.

Há ainda vários métodos baseados em simulações de processos, porém não são voltados especificamente para a validação dos processos de negócio em relação às regras de negócio (JANSEN-VULLERS; NETJES, 2006; BARJIS; SHISHKOV; DIETZ, 2001; KELLNER; MADACHY; RAFFO, 1999). Outra abordagem é proposta por Denis (2009). Este método utiliza os diagramas de atividades da UML expressando as regras com OCL. Com isso é gerado um grafo de fluxo de fluxo que representa o processo com suas regras de negócio. Para a validação é realizada uma animação com um cenário específico, determinado pelo usuário. Para persistir os objetos criados durante a animação, é usado um banco de dados relacionais. Então é necessário

o mapeamento do modelo conceitual de informação para as entidades relacionai e o mapeamento das regras de negócio e das ações das atividades para instruções SQL.

1.5 Organização da Dissertação

Este trabalho está dividido em um total de 6 capítulos. Este primeiro faz uma contextualização e uma introdução sobre o problema que o motivou, descrevendo a motivação, os objetivos a serem alcançados e suas contribuições. O capítulo 2 faz uma revisão da literatura necessária para o entendimento deste trabalho. No capítulo 3 é descrita a solução para o problema apresentado, explicando todos os passos para se realizar a validação das regras de negócio com o processo de negócio. O capítulo 4 apresenta a ferramenta PRUV, desenvolvida para dar apoio e automatizar o método. O capítulo 5 mostra um exemplo de como o método é utilizado na prática. O capítulo 6 trata das considerações finais a cerca deste trabalho.

2 REVISÃO DE LITERATURA

Para entender o funcionamento de uma organização é importante que sejam identificadas as regras de negócio. Uma vez identificadas, o conhecimento expresso nessas regras traz uma vantagem competitiva, além de poder auxiliar no processo do desenvolvimento de sistemas de informações, já que os sistemas que apoiam as organizações precisam ter as regras de negócio embutidas (ROSCA; GREENSPAN; WILD, 2002). Outro fator cujo conhecimento é de extrema importância dentro de uma organização são seus processos de negócio, já que estão dispersos pelas diversas áreas funcionais. A efetiva compreensão dos processos pode ser auxiliada através da sua modelagem (BARBALHO; ROZENFELD; AMARAL, 2002).

O método que apresentaremos aqui tem a finalidade de validar a conformidade de processos de negócio com as regras de negócio através da simulação de cenários do processo. Para isso é necessário a elaboração e integração de dois modelos: o modelo de regras de negócio e o modelo de processo. Este capítulo apresenta os principais conceitos envolvidos na construção de cada um desses modelos.

A Unified Modeling Language (UML) está se tornando o principal padrão para a especificação de estrutura, comportamento e arquitetura de software e caminha para

especificações cada vez mais consistentes acerca da modelagem de processos de negócio (BOOCH; RUMBAUGH; JACOBSON, 2005). A UML abrange vários tipos de diagramas, cada um destinado a representar uma diferente visão do sistema de informação. Exemplos de diagramas populares são: diagramas de casos de uso, diagramas de classe e interação, que descrevem, respectivamente, requisitos funcionais, e características estáticas e dinâmicas do sistema e o diagrama de atividades, que representa os fluxos conduzidos por processamentos (OMG, 2005).

A Object Constraint Language (OCL) é uma linguagem para representar restrições aos modelos UML. Vários trabalhos propõem o uso OCL de para representar regras de negócio. A utilização da OCL apresenta várias vantagens, já é uma linguagem bastante abrangente, sendo até mesmo equivalente à Structured Query Language (SQL)[1] em poder de computação, sendo, porém, é mais clara e concisa (VARELLA et al., 2004; ZIMBRÃO et al., 2002).

Além disso, a OCL pode ser usada tanto por analistas de sistemas, que têm uma melhor compreensão de linguagens de programação, quanto por analistas do negócio, que têm um conhecimento mais limitado neste sentido. Isso porque existem vários esforços no sentido de transformar linguagens mais próximas das naturais em OCL, como nos trabalhos de Cunha (2009) e Varella (2004). Também há esforços no sentido contrário, de trazer a OCL para linguagem natural, tendo como exemplo a ferramenta FalaOCL (ZIMBRÃO et al., 2002).

[1]SQL, ou Linguagem de Consulta Estruturada, é uma linguagem de pesquisa declarativa para banco de dados relacional (base de dados relacional).

2.1 Modelo de Regras de Negócio

Regras de Negócio (RN) são restrições que definem condições que devem ser verdadeiras em determinadas situações. RNs não são descrições de um processo, elas definem condições sob as quais um processo deve ser realizado ou as novas condições que existirão depois que um processo tenha sido concluído (MORGAN, 2002).

A execução de um processo de negócio exige a criação, acesso, atualização e exclusão de objetos de negócios. De acordo com Morgan (2002), "objetos de negócio são todas as coisas com as quais o negócio tem que lidar". Além disso, nós estamos assumindo que este termo se refere a categorias de coisas, chamadas classes, que podem ser concretas ou abstratas e são comumente usadas durante operações de negócios.

Um modelo de classes representa os objetos do negócio, suas características e relacionamentos em um formato independente de implementação (LONGLEY et al., 2007). Um modelo de classes é resultado de uma classificação imposta a um conjunto de objetos presentes em um ambiente devido às suas similaridades. Os objetos representados que são instâncias desses tipos de classes são chamados simplesmente de objetos de negócio. O diagrama de Classes de Domínio é uma especificação para termos de negócio e seus relacionamentos. Usando esse diagrama, é possível navegar de uma classe, ou termo, para seus relacionamentos (OMG, 2005). A figura 2.1 mostra o metamodelo definido pela OMG para descrever a estrutura do diagrama de classes da UML utilizando o *Metamodel Object Facility* (MOF)[2].

Regras de Negócio precisam ser traduzidas para um formato computacional antes de poderem ser aplicadas a qualquer sistema de informação. Isto significa que as regras devem ser expressas como restrições em um modelo de classes de domínio que

[2]MOF se originou na UML e é padrão internacional para a escrita de metamodelos: ISO/IEC 19502:2005 Information technology – Meta Object Facility (MOF)

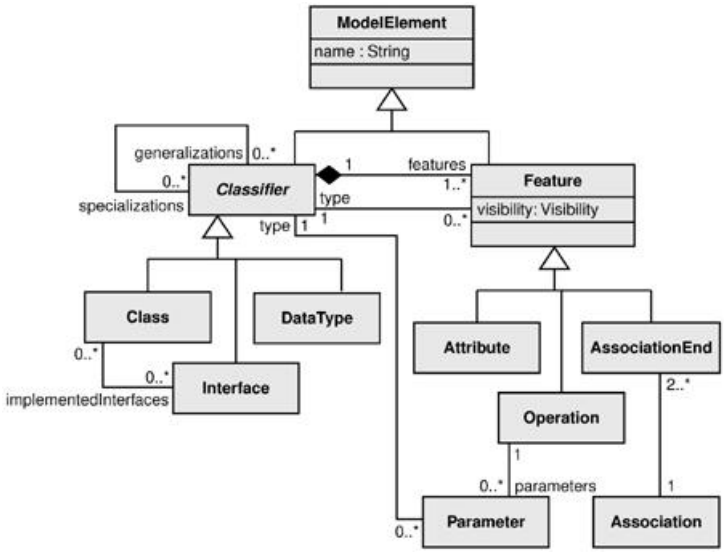


Figura 2.1: Metamodelo do Diagrama de Classes da UML (OMG, 2005)

incorpore todos os termos e fatos do vocabulário de negócio (CUNHA, 2009). Já que as sentenças de regras de negócio precisam estar em um formato compreensível ao dono do negócio, o requisito para um formato computacional irá requerer o envolvimento tanto da equipe de tecnologia da informação quanto o uso de ferramentas automáticas de tradução, que irão traduzir as especificações de regras de negócio para formatos computacionais que possam ser usados nos sistemas de informação do negócio.

O trabalho de Cunha (2009) transforma os padrões da Semantics of Business Vocabulary and Business Rules (SBVR)[3] em OCL. A SBVR permite a expressão de regras de negócio utilizando linguagem natural controlada, enquanto a OCL é uma linguagem computacionalmente manipulável.

Regras de Negócio podem estar presentes tanto no modelo de classes de domínio quanto no modelo de processo. No método apresentado neste trabalho, as regras

[3]A SBVR é o padrão do OMG para a construção de vocabulários e regras de negócio no nível do modelo independente de computação (CIM) da MDA.

representam, no modelo de classes de domínio, restrições aos valores dos objetos e aos relacionamentos incluídos em tal modelo. Já no modelo de processo, elas definem o que deve acontecer ao invés de especificar como deve acontecer.

2.1.1 Object Constraint Language

A OCL, ou linguagem de restrição de objetos, que está presente na UML a partir da versão 2.0 (OMG, 2005), é a linguagem usada para a representação de regras de negócio neste método. É usada para expressar restrições em modelos orientados a objetos. Ela complementa os aspectos de diagramação da UML, permitindo ao modelador expressar coisas que o diagrama não permite totalmente ou esclarecer ambiguidades que geralmente surgem quando diagramas de classes UML tentam mostrar relacionamentos complexos (KLEPPE; WARMER; BAST, 2003). A OCL é uma linguagem declarativa que pode ser usada para especificar restrições, definidas como expressões, em modelos orientados a objetos. Essas expressões são puramente declarativas e sem efeitos colaterais, o que significa que uma expressão OCL não pode mudar o valor de qualquer objeto no modelo (OMG, 2003).

Cada restrição é definida em um contexto, que geralmente é associado a um termo do negócio. Sentenças em OCL são construídas em quatro partes: um contexto, que define a situação limitada na qual a sentença é válida; uma propriedade, que representa algumas características do contexto (por exemplo, se o contexto é uma classe, uma propriedade pode ser um atributo); uma operação (por exemplo, aritmética, em conjuntos), que manipula ou qualifica uma propriedade, e palavras-chave (por exemplo, *if, then, else, and, or, not, implies*), que são usadas para especificarem expressões condicionais (OMG, 2003). A figura 2.3 mostra o metamodelo da OCL, que reutiliza algumas classes do metamodelo da UML e é definido usando a linguagem de modelagem MOF.

A OCL foi definida com o objetivo de ser uma linguagem de uso mais fácil, porém, especificações escritas nesta linguagem podem apresentar problemas de legibilidade e manutenibilidade. Em seu trabalho, Correa (2006) propõe um conjunto de reestruturações com o objetivo de apoiar a substituição dessas construções por outras mais adequadas. Propõe também uma abordagem e desenvolve um software para apoiar a automação de reestruturações e a verificação da preservação da semântica do modelo no caso de reestruturações realizadas manualmente.

No exemplo a seguir temos um trecho de um modelo de classes de uma locadora de carros na figura 2.2 e também algumas restrições escritas em OCL:

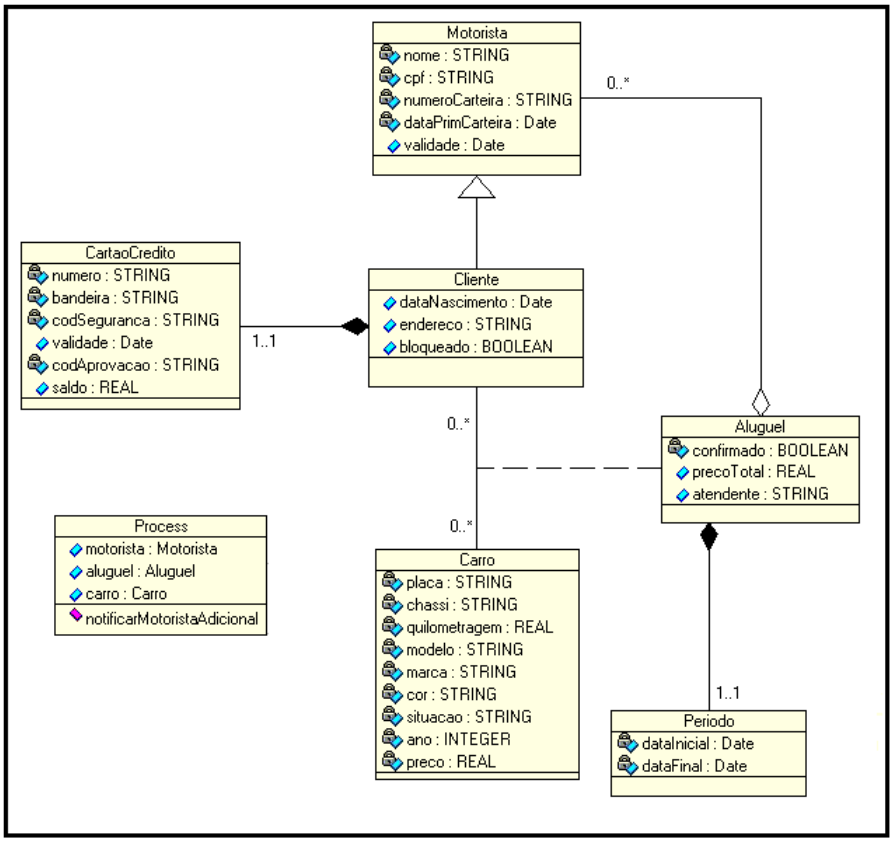


Figura 2.2: Modelo de classes de uma locadora de carros

```
context Process::notificarMotoristaAdicional(umMotorista: Motorista)
```

```
pre pre1: self.aluguel.motoristaAdicional->size() < 2
post post1: self.aluguel.motoristaAdicional@pre->size() = 0 implies
  self.aluguel.precoTotal = self.aluguel.precoTotal@pre * 1.25
post post2: self.aluguel.motoristaAdicional->including(umMotorista)

context Carro
inv CarroPlacaUnica: Carro.allInstances->isUnique(placa)
inv CarChassiUnico: Carro.allInstances->isUnique(chassi)
```

Neste exemplo temos trechos de regras de um processo de aluguel de carros. A pré-condição da operação *notificarMotoristaAdicional* diz que a quantidade de motoristas ligados a um aluguel através do relacionamento *motoristaAdicional* deve ser menor que 2. A pós-condição diz que se a quantidade de motoristas do relacionamento *motoristaAdicional* antes da operação *notificarMotoristaAdicional* ser executada é zero implica que o atributo *aluguel.precoTotal*, depois da execução da operação, deve estar multiplicado por 1,25.

A invariante *CarroPlacaUnica* diz que o atributo *placa* da classe *Carro* deve ser único em todas as suas instâncias. A invariante *CarChassiUnico* faz a mesma restrição para o atributo *chassi*.

2.1.2 A Snapshot Sequence Language

Como já foi dito anteriormente, a OCL é uma linguagem puramente declarativa, sem efeitos colaterais, ou seja, não é possível realizar nenhuma alteração no estado dos objetos de um modelo utilizando OCL. A Snapshot Sequence Language (ASSL) (RICHTERS; GOGOLLA, 2001) é uma linguagem que define cenários de objetos e descreve como criá-los, além de permitir verificar propriedades sobre modelos UML.

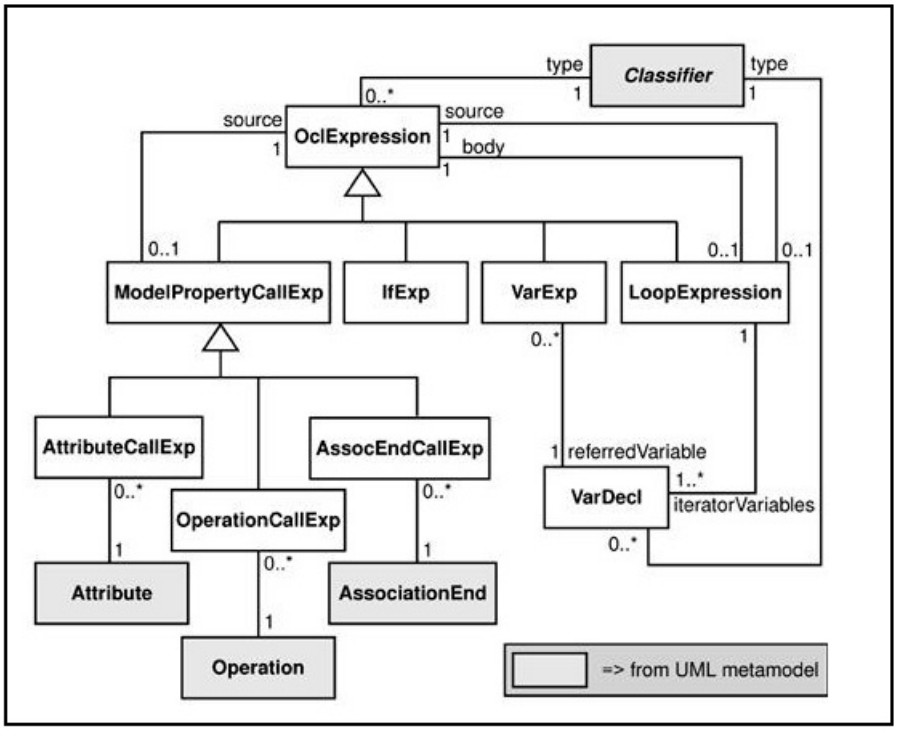


Figura 2.3: Metamodelo da OCL (OMG, 2003)

Ela surgiu no intuito de realizar manipulações nos objetos de um modelo de classes e se utiliza também de OCL (GOGOLLA; BÜTTNER; RICHTERS, 2005). Com essa linguagem é possível criar e deletar objetos, que são instâncias das classes, criar e deletar associações entre objetos e modificar os atributos dos objetos através de sequências de comandos.

Exemplo de procedimento escrito em ASSL, utilizando novamente o diagrama de classes da figura 2.2:

```
procedure verificarCredito(umCartao: CartaoCredito, umCliente: Cliente)
begin
  [umCartao].codAprovacao := ['aprovado'];
  Insert(CartaoCredito_Cliente, [umCliente], [umCartao]);
```

```
end;
```

Nos procedimentos ASSL, todos os valores e variáveis são expressões OCL, por isso devem ser usados entre colchetes. O único caso em que não se usa colchetes são variáveis ou atributos recebendo valores. No exemplo acima, o procedimento verificarCredito(umCartao: CartaoCredito) tem como entrada as variáveis *umCartao*, que é do tipo *CartaoCredito* e *umCliente*, que é do tipo *Cliente* (essas classes devem estar presentes no modelo de classes). O comando [umCartao].codAprovacao := ['aprovado'] atribui ao atributo *codAprovacao* da variável *umCartao* o valor *'aprovado'*. O comando Insert(CartaoCredito_Cliente, [process.cliente], [umCartao]) cria uma ligação com o nome *CartaoCredito_Cliente* entre os objetos *umCliente* e *umCartao*.

A ASSL é um complemento para a ferramenta UML-Based Specification Environment (USE)[4] (GOGOLLA; BÜTTNER; RICHTERS, 2005). A USE auxilia desenvolvedores na análise da estrutura de modelos e seu comportamento através da geração de cenários, que são obtidos através de uma sequência de comandos e operações. Os usuários podem verificar formalmente invariantes, pré e pós-condições de acordo com o comportamento esperado, além de possuir outras funcionalidades (GOGOLLA; BÜTTNER; RICHTERS, 2007).

No procedimento ASSL descrito acima, se a atividade tivesse como saída os objetos *umCartao: CartaoCredito* e *umCliente: Cliente*, bastaria adicionar a seguinte linha de código ao procedimento: return umCartao, umCliente;.

[4] A ferramenta USE, sua documentação e seu código-fonte e a especificação da linguagem ASSL podem ser encontrados no endereço http://www.db.informatik.uni-bremen.de/projects/USE/

2.2 Modelo de Processo

A representação de um processo de negócio deve mostrar todas as suas características importantes. O modelo de processo deve conter informações sobre: atividades, pessoas, funções, dados, unidades organizacionais, produtos, sistemas de informações e regras de negócio (MORGADO, 2007). Isso tem sido um assunto de várias publicações na literatura especializada. Um levantamento interessante sobre esse assunto pode ser encontrado no artigo de Sadiq, Governatori e Namiri (2007), onde os autores apresentam as principais técnicas para a modelagem de processos de negócio, cobrindo objetivos, notação e a linguagem usada no processo de modelagem.

Há uma grande variedade de métodos e notações que podem ser usados para a descrição de processos. Essa gama de métodos vai desde notações gráficas fáceis de entender até rigorosos formalismos matemáticos. Uma das principais vantagens de notações formais é que através delas é possível realizar simulações de computador e, então, permitir um estudo detalhado do comportamento do processo (MUEHLEN; INDULSKA; KAMP, 2007). Por outro lado, notações formais são difíceis para pessoas não técnicas entenderem. A falta de um entendimento pleno torna extremamente difícil a tarefa de validar a execução de cenários com os usuários dos processos.

Notações gráficas são excelentes para apresentação e discussão, mas como elas não têm o rigor de linguagens formais, isso torna difícil a obtenção de dados quantitativos mais precisos sobre o comportamento do processo. Face à sua simplicidade, este é o método preferido na vasta maioria de projetos de modelagem de processos de negócio (HALVEY; MELBY, 2007). Não há uma técnica de modelagem de processos compreensível que seja amplamente aceita. Tanto pessoas da área acadêmica quanto do mercado de trabalho têm usado um conjunto de ferramentas para modelagem de processos, misturando métodos, notações e ferramentas dependendo do problema

em consideração (SADIQ; GOVERNATORI; NAMIRI, 2007).

A UML também pode ser usada para descrever modelos de negócio. O Diagrama de Atividades (DA) da UML é uma notação de modelagem simples e efetiva que pode ser usada para descrever desde simples atividades até fluxos de trabalho complexos (OMG, 2005). DAs enfatizam o fluxo de controle entre as atividades. Além disso, já que tanto os recursos necessários quanto os produtos gerados têm uma representação explícita, DAs podem descrever a dinâmica de processos de negócio (BOOCH; RUMBAUGH; JACOBSON, 2005). Sendo assim, DAs são funcionalmente similares a Redes de Petri (DESEL; JUHÁS, 2001), grafos de fluxo (POOLE, 1995) e Event-driven Process Chains (EPC) (TSAI et al., 2006).

A especificação formal de diagramas de atividades é descrita pelo metamodelo de Diagrama de Atividades da UML 2.0 que, por sua vez, é expresso usando o MOF como sua linguagem de especificação, conforme mostra a figura 2.4 (OMG, 2005). Conceitualmente um DA é um tipo de grafo direcionado, com nós e arestas, chamadas transições. Na transição $t = (v, w)$ de um DA, o nó w é um vizinho de saída, e t uma transição de saída de v; o nó v é um vizinho de entrada, e t uma transição de entrada de w. A seguir apresentamos os principais elementos de um DA.

Um nó pode ser especializado em três tipos:

1. nó de ação, que realiza operações nos valores que recebe;
2. nó de controle, que define o fluxo de execução das atividades;
3. nó objeto, que representa os dados usados na execução das atividades.

Um nó de controle pode ser especializado em seis tipos de nós:

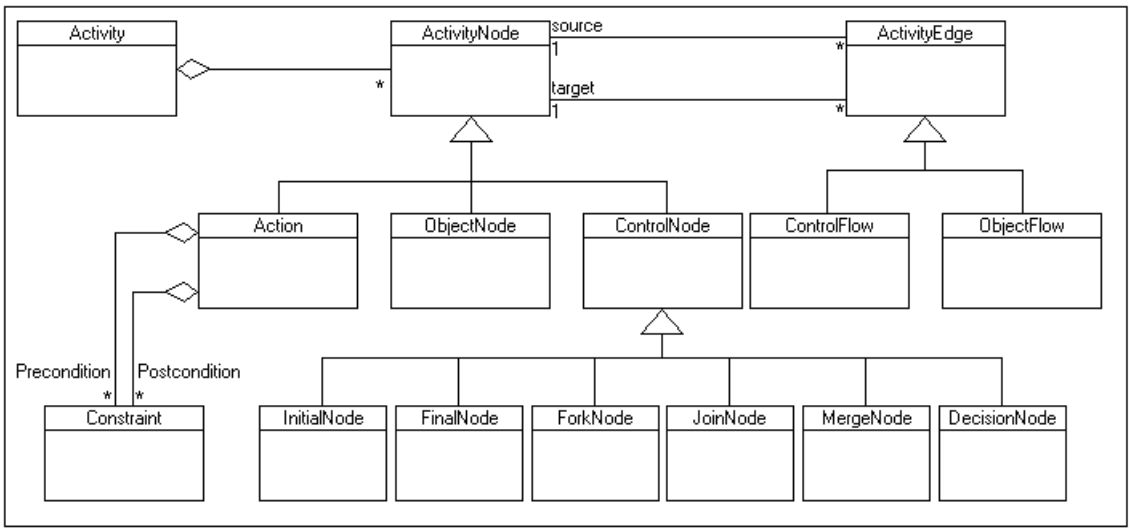


Figura 2.4: Metamodelo do Diagrama de Atividades da UML (OMG, 2005)

1. nó Inicial, que marca o início do processo;
2. nó Final, que marca o fim do processo;
3. nó Decisão, que representa um ponto na execução do processo onde o fluxo de trabalho segue um entre vários fluxos disjuntos;
4. nó Merge, que pode ser visto como um *OU* lógico e representa um ponto onde os fluxos de trabalho convergem;
5. nó Fork, que simboliza o início concorrente de vários fluxos de trabalho;
6. nó Join, que pode ser visto como um *AND* lógico, que representa o sincronismo de vários fluxos de trabalho.

O nó de ação é representado pela Atividade. Atividades são tarefas que precisam ser executadas para se alcançar determinado objetivo. Atividades gastam tempo e recursos para serem executadas e podem gerar produtos (MORGADO, 2007). Um modelo de processo deve representar os diferentes objetos que as atividades usam

durante a execução. Esses objetos são representados pelos nós Objeto e podem ser recursos consumidos ou produtos gerados.

Transições constituem as arestas do grafo direcionado. Elas representam a ligação dos nós necessários para a execução do fluxo de trabalho. Uma transição pode ser acionada por um evento, habilitada por condições de guarda e iniciar a execução de uma ação. Uma Condição de Guarda (CG) pode representar tanto uma alternativa de uma decisão ou pós-condições de execução de uma atividade (OMG, 2005). Uma transição pode ser especializada em fluxo de controle, que são ligações entre nós de ação ou controle e fluxo de objetos, que são ligações entre atividades e objetos.

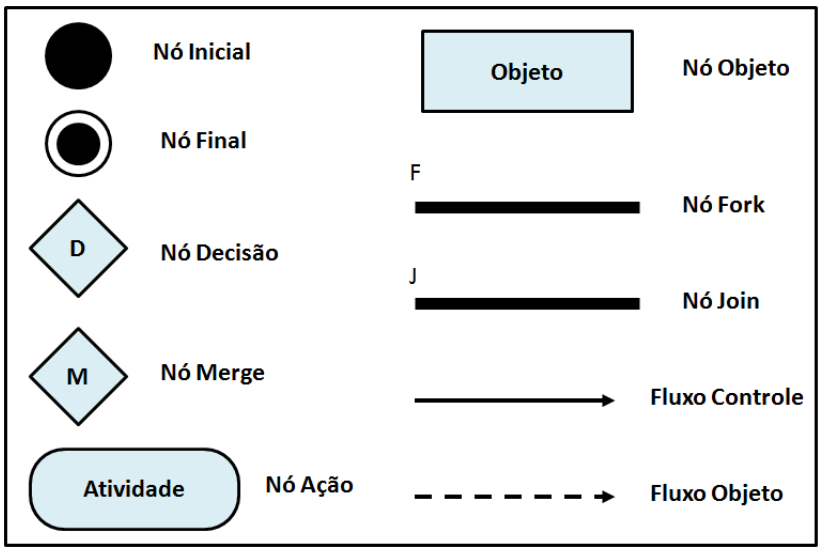


Figura 2.5: Elementos principais do Diagrama de Atividades UML

2.3 Simulação de Processos de Negócio

Simulação é o processo de modelar um sistema real ou imaginário e realizar experimentos neste modelo com a finalidade de entender seu comportamento ou avaliar estratégias para a operação do sistema, melhora de desempenho, testes, treinamen-

tos e educação. Simulação se tornou uma técnica amplamente usada para elicitação e validação de requisitos de *software* (SOKOLOWSKI; BANKS, 2009). Também pode ser usada para dar suporte à validação de processos de negócio (JANSEN-VULLERS; NETJES, 2006). A simulação disponibiliza um retorno da interação entre usuários e especialistas de negócio que não pode ser fornecido por outras técnicas, como prototipagem (BARJIS; SHISHKOV; DIETZ, 2001). Ela não só permite uma ligação direta com a especificação do processo, mas também é, por concepção, adequada para execução repetitiva (BARJIS, 2007). Esta característica acaba sendo muito importante para verificar o comportamento do processo depois que mudanças são introduzidas.

Simulação pode ser usada para auxiliar em decisões de melhorias de processo, ajudando a prever o impacto que uma mudança pode causar em um processo antes de colocar essa mudança em prática. Animação é um tipo de simulação que ajuda as pessoas a visualizar questões relacionadas ao fluxo dos processos (KELLNER; MADACHY; RAFFO, 1999).

Por outro lado, da mesma maneira que abordagens orientadas a teste, o objetivo da simulação é encontrar erros. Uma simulação não pode provar que o modelo de processo é completamente compatível com as regras de negócio, mas pode ser usada a fim de se obter um nível de confiança suficientemente satisfatório na qualidade de especificação do processo (LALIOTI, 1997). A técnica de simulação tenta encontrar um equilíbrio entre a completude, velocidade e facilidade de uso. Quando adequadamente automatizada, provê uma resposta rápida e uma menor dependência do uso de ferramentas de validação formal que, apesar de fornecerem maior precisão, têm um custo de trabalho que exige o conhecimento de métodos formais (PEREIRA, 2009).

Kellner, Madachy e Raffo (1999) focam seu trabalho na simulação de processos de

software, mas afirmam que os conceitos apresentados podem ser extendidos para os demais tipos de simulação. Eles respondem às questões: porque, o quê e como simular. Além disso apresentam algumas técnicas de simulação.

Modelos para simulação podem ser visuais ou textuais. Os visuais têm se tornado um padrão para simulação, devido à sua facilidade de desenvolvimento e também porque possibilitam a animação do modelo sendo simulado (KELLNER; MADACHY; RAFFO, 1999). Os autores citam dez diferentes técnicas e focos de simulação, mas nenhuma delas é voltada para a validação de regras e processos de negócio.

Jansen-Vullers e Netjes (JANSEN-VULLERS; NETJES, 2006) classificam ferramentas de simulação de processos em três categorias:

- ferramentas de modelagem de processos de negócio,
- ferramentas de gerenciamento de processos de negócio,
- ferramentas de simulação de propósito geral.

As ferramentas de modelagem de processos de negócio servem para descrever e analisar os processos. Nessa categoria há a ferramenta Petros, que utiliza redes de Petri como modelo. Seu objetivo é definir modelos de processos de negócio visando a qualidade de sistemas de gerenciamento, reespecificação dos processos de negócio, melhoria de comunicação entre as partes interessadas ou a implementação de sistemas de gestão de fluxos de trabalho (JANSEN-VULLERS; NETJES, 2006).

Outra ferramenta citada é a Aris, baseada em EPCs. É uma ferramenta profissional para a análise dinâmica de processos de negócio. Sua simulação mostra se o processo pode ser executado por completo e dá informações sobre tempos de processamento e nível de utilização de recursos (JANSEN-VULLERS; NETJES, 2006).

As ferramentas de gerenciamento de processos de negócio servem para automatizar o fluxo do trabalho e apoiar todo o ciclo de vida do processo. As ferramentas de simulação desse tipo não visam a validação de processos de negócio. As ferramentas de simulação de propósito geral são ferramentas não adaptadas para um domínio de negócio específico, mas são para uso em negócios em geral. Dentre as ferramentas conhecidas, as simulações servem para análise de dados estatísticos dos processos, dentre outras coisas, mas sem o propósito de validação (JANSEN-VULLERS; NETJES, 2006).

O método apresentado por Barjis (2001) apresenta processos de negócio em termos de componentes de negócio e tem como objetivo validar o comportamento de tais componentes. Considera o diagrama de atividades da UML como uma ferramenta de pré-simulação e uma rede de Petri como entrada para a simulação. Dessa forma, este método exige conhecimentos de UML, componentes de negócio e redes de Petri.

O método apresentado nesta dissertação conta com a etapa de simulação do processo para que se faça a validação no modelo do processo com as regras de negócio. Com a ferramenta desenvolvida para apoiar o método é possível ter estatísticas sobre qual a cobertura do processo alcançada com as simulações. Além disso, ele conta com a vantagem de utilizar apenas UML para modelagem e ASSL como complemento, o que faz com que pessoas com menor conhecimento técnico consigam usá-lo.

2.4 Abordagens para a Geração de Cenários de Teste

Há muitas abordagens para validar modelos. Uma delas é a validação estática, que pode ser feita por revisões ou inspeções, entre outros (RIBEIRO, 2009). Outra abordagem é a validação dinâmica, que consiste na simulação da execução do modelo. Isso exige a geração de um conjunto de casos de testes, ou cenários (KHARBILI et al.,

2008). Cada cenário de teste gera uma instância do processo com todos os objetos de negócio envolvidos, correspondendo a uma simulação de uma situação real na qual o processo seria executado.

A geração de um conjunto de cenários de teste é análoga à seleção de conjuntos de casos de teste para programas de computador, no sentido de que ambos usam a representação de grafo de controle como base para a definição do conjunto de casos de teste. A diferença é que conjuntos de casos de teste são criados para encontrar erros na implementação de um programa, e o objetivo da simulação de um cenário é detectar situações de não conformidade de instâncias do modelo de processo com as regras de negócio. A geração de cenários de teste é alvo de vários estudos e possui várias abordagens. A seguir, listamos e explicamos brevemente algumas dessas abordagens.

A técnica chamada Scatter Search baseia-se em algoritmos genéticos, que busca a evolução de um conjunto de soluções encontradas a cada iteração. Essas soluções são armazenadas e combinadas para serem melhoradas através de um método de refinamento, a fim de se chegar a soluções que se ajustam melhor que as já encontradas. Para determinar se a solução é boa, são analisados seu custo e sua diversidade. Seu objetivo é cobrir o maior número possível de nós em um grafo direcionado. Um conjunto bem diversificado de soluções permite um conjunto de busca maior (BLANCO; TUYA; ADENSO-DÍAZ, 2009).

Apesar desse método ser bem sofisticado e buscar a maior cobertura de um grafo direcionado, este trabalho gera apenas números. Outra limitação é que seu algoritmo de refinamento e geração de cenários baseia-se apenas nas regras ou restrições contidas diretamente nos nós de decisão, e não leva em conta demais regras, ignorando invariantes e pré e pós-condições.

No nosso trabalho necessitamos gerar casos de testes a partir de uma especificação formal, neste caso OCL. O problema de gerar casos de testes através de especificações formais é um problema de satisfação de restrições (Constraint Satisfaction Problem - CSP). Um CSP é formado por um conjunto finito de variáveis e restrições. Cada variável está associada a um domínio, ou seja, um conjunto de possíveis valores. Então, uma restrição é uma relação entre as variáveis, formando combinações válidas para os valores destas variáveis. Uma solução para um CSP é um conjunto de valores para cada variável, dentro de seu domínio, tal que todas as restrições sejam satisfeitas (SALAS; AICHERNIG, 2005).

Segundo Chen, Mishra e Kalita (2008), cenários de teste gerados a partir de diagramas de atividades da UML não só podem ser usados para garantir a consistência entre níveis de abstração mas também podem ser reusados para reduzir o custo total da validação de sistemas. Porém, ainda não há técnicas suficientes para a geração automatizada de testes dirigidos a partir de diagrama de atividades da UML.

Ainda em (CHEN; MISHRA; KALITA, 2008) é proposta uma abordagem para a geração automática de testes dirigidos a partir desses diagramas utilizando verificação de modelos. Os autores definem a especificação de cobertura dos diagramas de atividades UML para gerar propriedades necessárias, propõem um conjunto de regras para transformar os diagramas de atividades em um modelo formal e realizar a verificação no modelo formal e nas propriedades geradas. Eles ainda definem três critérios para a cobertura do diagrama de atividades:

Cobertura de Atividade Todas as atividades do diagrama precisam ser cobertas (visitadas). O valor da cobertura de atividades é a proporção entre as atividades cobertas e o número total de atividades do diagrama.

Cobertura de Transição Todas as transições do diagrama precisam ser cobertas

(visitadas). O valor da cobertura de transição é a proporção entre as transições visitadas e o número total de transições do diagrama.

Cobertura de Caminho-Chave Um caminho-chave, também chamado de caminho básico ou caminho independente, é um caminho em um fluxo de controle, neste caso, um diagrama de atividades, onde pelo menos um nó ou uma transição contidos neste caminho não estão contidos em nenhum outro caminho-chave (PRESSMAN, 2006). Todos os caminhos-chave do diagrama precisam ser cobertos (visitados). O valor da cobertura de caminho-chave é a proporção entre os caminhos-chave visitados e o número total de caminhos-chave do diagrama.

A princípio, o diagrama de atividades da UML é traduzido para um modelo formal, no formato NuSMV (CIMATTI et al., 2002), depois as propriedades necessárias são geradas de acordo com o critério de cobertura desejado. Por último, uma versão negada dessas propriedades são aplicadas ao modelo formal para gerar os testes, a fim de se encontrar contra-exemplos. Apesar de promissora, essa técnica é complexa e pode levar a uma explosão no espaço de testes, caso o diagrama seja consideravelmente complexo, levando, assim, muito tempo para sua execução.

O trabalho de Salas e Aichernig (2005) baseia-se em mutações para a geração automática de casos de teste para OCL para antecipar erros que poderiam vir a acontecer. Esses erros podem ser antecipados gerando-se uma quantidade suficiente de casos de testes. Os erros são detectados através da mutação das pré e pós-condições em OCL, e não apenas a cobertura dos testes é analisada, mas também são gerados testes que cobrem as falhas (mutações) introduzidas. Os autores buscam, em sua abordagem, resolver CSPs criando uma relação entre testes baseados em falhas e resolução de restrições. Então, ao invés de focarem na cobertura da estrutura da especificação, que é possivelmente diferente da estrutura da implementação, eles buscam as fa-

lhas. Porém, algumas falhas não podem ser antecipadas e, portanto, não podem ser testadas.

Weißleder e Sokenou (2008) fazem um breve esboço sobre algumas abordagens que utilizam UML e OCL. Citam uma abordagem que relaciona as pré e pós condições de uma operação e deriva informações comportamentais a partir de um modelo estático. Outra abordagem utiliza o OCL como um oráculo adicional para testes, onde as expressões OCL são apenas transformadas numa outra linguagem formal. Um trabalho mais avançado que suporta um subconjunto da OCL e utiliza informações de fluxo de controle de uma máquina de estados para combinar condições OCL, onde equações de pós-condições são interpretadas como assinaturas, a fim de permitir que o modelo tenha uma execução simbólica.

Weißleder e Schlingloff (2007a; 2007b) lidam com geração automática de casos de testes baseados em modelos através do uso de modelos casados. Os modelos consistem de diagramas de máquinas de estados da UML, diagrama de classes e expressões OCL(pré e pós-condições de operações e condições de guarda da máquina de estados) nesses dois diagramas. Condições de guarda podem ser afetadas por pós-condições de uma transição anterior a estas, com isso pode-se achar uma relação entre elementos de condições de guarda e pré-condições e elementos de pós-condições. Partições de intervalos de valores são derivadas automaticamente através da avaliação de expressões OCL para testes de fronteira, que podem ser usados para encontrar erros correspondentes a diferenças entre as restrições no modelo e as restrições no sistema em teste. A qualidade dos testes gerados depende dos critérios de cobertura e da escolha correta da partição de fronteiras. As condições de guarda podem ser transformadas em condições para parâmetros de entrada de operações, e essas condições são interpretadas como partição de fronteiras. Os valores de fronteira são gerados automaticamente a partir das expressões OCL.

A princípio as interdependências entre as expressões são analisadas, depois o modelo é transformado numa árvore de transições, e os caminhos de transições da árvore são investigados. Em relação a outras abordagens, este trabalho tem como contribuição um método de geração de casos de testes para validar expressões OCL em pós-condições. Essa abordagem permite combinar critérios como baseados em cobertura de fronteiras com critérios baseados em cobertura de fluxo de controle. Além disso, a ferramenta criada pelos autores é a primeira a gerar classes de equivalência para parâmetros de entradas de testes a partir de UML e OCL (WEIßLEDER; SOKENOU, 2008).

2.5 Abordagem para a Geração do Conjunto de Cenários de Teste

A abordagem elaborada por Weißleder e Schlingloff (2007a; 2007b) mostrou-se mais adequada para ser usada nesse trabalho, pois lida diretamente com diagramas UML e utiliza de maneira mais efetiva e ampla as restrições OCL. Além disso, o foco dos testes é gerar valores de fronteira para validar as pós-condições das restrições em cada estado do sistema.

As técnicas de teste de partição e de fronteira são bem conhecidas e normalmente são usadas juntas, já que particionar as entradas dos testes em valores de domínios é um pré-requisito para testes baseados em fronteiras de domínios. São usados em casos onde os valores exatos das fronteiras são importantes, como por exemplo, medidas de objetos, posição, idade, etc. Desta forma, trazendo essa abordagem para realizar a validação das regras de negócio com os processos de negócio, os cenários de teste são gerados com foco nas restrições, que são as regras de negócio, e utilizados para realizar a simulação dos processos.

2.5.1 Classificação de Variáveis das Expressões OCL

Para a técnica de partições e de fronteira é necessário classificar as variáveis envolvidas nas expressões OCL para que seja possível analisar estaticamente a interdependência dessas expressões dentro do sistema. Após feita essa classificação e analisada a interdependência das variáveis é possível realizar transformações nas expressões OCL, derivar partições de valores e gerar valores de entrada concretos para os testes a partir das partições.

Os predicados das expressões OCL são classificados primeiramente em passivos ou ativos, tal que predicados ativos podem alterar o valor de atributos, enquanto os predicados passivos apenas podem ler tais atributos. Além disso, há uma classificação para variáveis também, para reconhecer quais podem ser alteradas e quais não podem. Toda a classificação descrita a seguir está de acordo com o trabalho de Weißleder e Schlingloff (2007a; 2007b).

A unidade de classificação atômica é *var*. Ela é parte de um predicado atômico, que é o *predicado contexto* de *var*. Cada predicado é formado por variáveis e relações e operações entre as variáveis. Atributos, parâmetros de entrada e constantes são as variáveis do sistema e podem ser *dependentes* ou *independentes*.

> **Definição 1 (Variáveis Dependentes e independentes).** Uma *variável independente* é tanto um parâmetro de entrada de um evento quanto um atributo constante de uma classe. Seu valor é constante. Uma *variável dependente* é um atributo não-constante de uma classe. (WEIßLEDER; SCHLINGLOFF, 2007a,b)

Uma variável *var* pode ser ativa ou passiva, dependendo de seu predicado contexto. Se uma variável está presente em uma pós-condição e não está ligada a *@pre* (que indica o valor da variável antes da execução da operação), então essa variável pode ser alterada, ou seja, é *ativa*. Em qualquer outro caso, essa variável é *passiva*, seu

valor não pode ser alterado.

Tabela 2.1: Variáveis ativas e passivas (WEIßLEDER; SCHLINGLOFF, 2007a,b).

Tipo de Expressão	**Variável Dependente**	**Variável Independente**
Pós-Condição (sem @pre)	ativa	passiva
Pós-Condição (com @pre)	passiva	passiva
Qualquer outro tipo	passiva	passiva

Definição 2 (Transição Predecessora) Assuma uma transição *t1* do caminho-chave que contém uma variável dependente passiva *var*. Então, a transição *t2* do caminho-chave, predecessora de *t1* com respeito a *var*, é a transição predecessora de *t1* que é mais próxima de *t1* e contém *var* como uma variável ativa. O valor da variável ativa *var* na transição predecessora corresponde ao valor de *var* em *t1*. (WEIßLEDER; SCHLINGLOFF, 2007a,b)

Definição 3 (Variáveis Definidas) Variáveis independentes são *definidas*. Variáveis ativas São *definidas* se todas as variáveis restantes contidas no seu predicado contexto forem *definidas*. Cada variável dependente passiva *depvar* contida em uma condição *cond* usada em uma transição *t2* do caminho-chave é definida se, e somente se, a transição predecessora *t1* de *t2* com respeito a *depvar* existe e a variável ativa correspondente é *definida*. (WEIßLEDER; SCHLINGLOFF, 2007a,b)

Teorema 1 (Variáveis Redutíveis) Em um conjunto de condições de uma expressão, cada variável dependente pode ser reduzida a variáveis independentes. (WEIßLEDER; SCHLINGLOFF, 2007a,b)

De acordo com a prova do Teorema 1, fornecida pelos autores, todas as variáveis dependentes de um conjunto de condições definido dependem diretamente ou indiretamente de variáveis independentes.

2.6 Tamanho do conjunto de teste

Namin e James (2009) realizaram um estudo no qual eles relacionam três propriedades de conjuntos de testes: tamanho, cobertura estrutural e eficácia para encontrar falhas. Em todo seu trabalho, eles tratam de testes através de mutações e o sorteio aleatório de mutantes. Neste caso eles consideram eficácia como o número de programas mutantes encontrados.

Adicionar um caso de teste a um conjunto de testes faz com que este conjunto seja, pelo menos, tão eficaz quanto era antes, e talvez até mais eficaz. Ou seja, para um dado conjunto de testes B, subconjunto de A, A será pelo menos tão eficaz quanto este conjunto B. Os autores dizem ainda que um teste adicional aumenta também a cobertura que o conjunto de testes alcança. No nosso trabalho usamos o critério de cobertura estrutural, ou seja, geramos um conjunto de testes de tal forma que todos os nós e todas as transições presentes no diagrama de atividades sendo testado sejam cobertos. Assim, o ato de adicionar mais um teste a este conjunto não aumenta a cobertura, já que, de acordo com o critério de cobertura adotado, o conjunto original cobre 100% do diagrama.

Através de experimentos realizados em alguns programas, percebeu-se que não há uma relação linear entre essas três variáveis em questão. Eles chegam a duas conclusões: A primeira é que ao alcançar um alto nível de confiança com um critério de cobertura forte leva automaticamente a uma maior eficácia dos testes, sem ter necessariamente que aumentar o conjunto de testes; a segunda é que, por causa da relação entre quantidade de teste e cobertura, quando um conjunto de testes está sendo construído, conforme o tamanho do conjunto cresce, o tamanho torna-se a variável mais importante para se alcançar uma maior cobertura (NAMIN; ANDREWS, 2009).

2.7 Considerações

Devido às suas características já apresentadas neste capítulo, a UML mostrou-se adequada para a representação dos modelos de regras de negócio e modelos de processo utilizados neste método. As regras de negócio são escritas em OCL e estão espalhadas pelos diagramas de classes de domínio e de atividades. Este último, é também utilizado para representar o processo de negócio.

A linguagem ASSL proposta por Gogolla e Richters (2005) será usada, juntamente com a OCL, dentro dos diagramas de atividades para especificar as ações que uma atividade realiza dentro do processo. Além disso, a ferramenta USE (GOGOLLA; BÜTTNER; RICHTERS, 2007) será usada como base para uma ferramenta criada para automatizar o método proposto neste trabalho.

A abordagem desenvolvida por Weißleder e Schlingloff (2007a; 2007b) será usada neste trabalho como base para a fase de geração de casos de testes, já que seu foco são os diagramas UML. Além disso, lida de maneira bastante abrangente com as restrições OCL, sendo capaz de gerar valores específicos para cada tipo de teste e utilizando diferentes critérios de cobertura para garantir maior eficácia nos resultados.

Uma adaptação do método proposto por esses autores é necessária para gerar o conjunto de casos de teste deste trabalho. Neste caso, ao invés do uso de diagramas de máquinas de estados são usados diagramas de atividades, que representam os processos de negócio. A mesma árvore de transição obtida usando-se os diagramas de máquinas de estado também pode ser obtida utilizando-se diagramas de atividades, com as devidas adaptações, onde cada atividade é considerada como uma operação.

3 UM MÉTODO PARA A VALIDAÇÃO DE REGRAS DE NEGÓCIO

Neste capítulo é descrito um método que é capaz de validar a conformidade de processos de negócio com regras de negócio de maneira mais eficaz e que é de mais fácil entendimento que técnicas existentes – por exemplo, *State Chats* (MUTH et al., 1998), redes de Petri (AALST, 2003), álgebra de processos (SCHROEDER, 1999), π-calculus (YANG; ZHANG, 2003) e lógica temporal (ESHUIS; WIERINGA, 2001) – já que não necessita de profundos conhecimentos de métodos formais. Neste método, processos de negócio são modelados através de diagramas de atividades da UML e o modelo conceitual é modelado usando-se diagrama de classes UML, enquanto regras de negócio são representadas como pré e pós condições em OCL anexadas a atividades do processo ou como invariantes OCL associadas às classes do diagrama de classes. A validação do modelo é baseada na simulação de um conjunto de cenários, que são instâncias do processo.

3.1 Diagramas de Atividades UML Bem-Formados

Apesar da UML ser um padrão amplamente reconhecido e utilizado, ela é criticada por sua semântica imprecisa, permitindo interpretação subjetiva e, consequentemente, gerando dificuldades no desenvolvimento da modelagem formal de negócios (ESHUIS; WIERINGA, 2001). Mesmo se levarmos em conta apenas o conjunto limitado de símbolos UML utilizado em nosso método, muitas construções que são sintaticamente corretas podem ter mais de uma interpretação.

Como uma solução que busca evitar múltiplas interpretações, nós propomos a inclusão de um conjunto de Regras de Formação para remover possíveis ambiguidades do modelo de processo antes de executar a simulação. Todos os modelos de processos que obedeçam a este conjunto de regras puramente sintáticas é chamado de Modelo de Processo Bem Formado (MPBF):

MPBF 1 O nó Inicial tem zero transições de entrada e uma transição de saída (Fig. 3.1(a)).

MPBF 2 O nó Final não tem transições saída e tem apenas uma transição de entrada (Fig. 3.1(b)).

MPBF 3 Atividades têm exatamente uma transição de entrada e uma de saída (Fig. 3.1(c)).

MPBF 4 O nó Decisão tem uma transição de entrada e duas ou mais transições de saída (Fig. 3.1(d)). Além disso, suas condições de guarda devem ser disjuntas. Por exemplo, as condições [process.carro.quilometragem <= 50000] e [process.carro.quilometragem > 50000] são disjuntas, já que o conjunto de valores possíveis para cada uma delas nunca terá valores em comum. Caso as condições de guarda não sejam disjuntas, pode acontecer o caso em que duas

ou mais condições são avaliadas como verdadeiras, o que constitui um erro.

MPBF 4 O nó Merge tem duas ou mais transições de entrada e uma transição de saída (Fig. 3.1(e)).

MPBF 5 O nó Fork tem uma transição de entrada e duas ou mais de saída (Fig. 3.1(f)). Todos os fluxos originados de um fork devem se encerrar em um mesmo nó join.

MPBF 6 O nó Join tem duas ou mais transições de entrada e uma da saída (Fig. 3.1(g)). Todos os fluxos que chegam a um join devem ter sido originados a partir do mesmo nó fork.

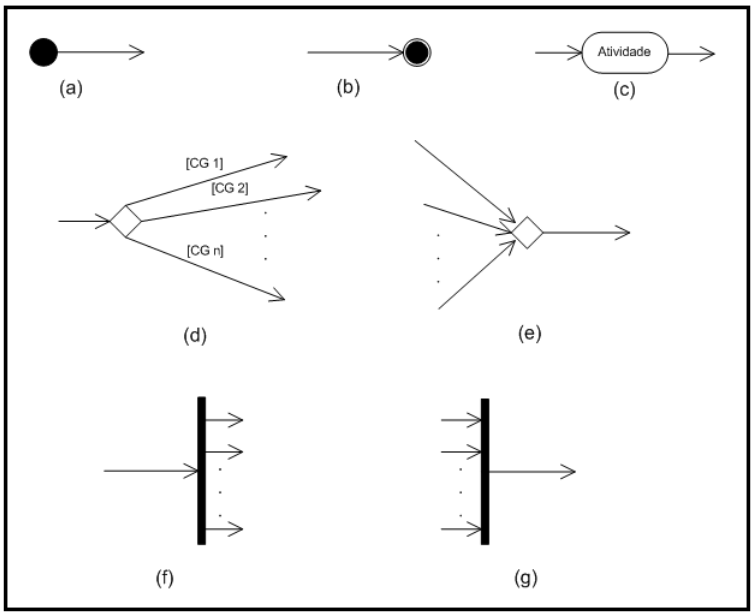


Figura 3.1: Nós bem formados

MPBF 7 Uma única atividade ligada a um nó Inicial e um Final é um MPBF (Fig. 3.2(a)).

MPBF 8 Uma conexão em série de MPBFs é um MPBF (Fig.3.2(b)).

MPBF 9 Um nó decisão conectado a MPBFs disjuntos e que terminam no mesmo nó merge é um MPBF (Fig. 3.2(c)).

MPBF 10 Um nó fork conectado a MPBFs disjuntos e que terminam no mesmo nó join é um MPBF (Fig. 3.2(d)).

3.2 Conformidade entre Processos e Regras de Negócio

Uma ação utiliza entradas e gera saídas seguindo uma sequência de passos pré-definida. Essa geração da saída é feita de acordo com um conjunto de restrições especificadas por pré-condições e pós-condições de ações. As pré-condições são condições ou predicados que devem ser verdadeiras imediatamente antes da execução de alguma operação em uma especificação formal. Elas definem restrições ao estados dos objetos pra que a execução ocorra com sucesso. Já as pós-condições são condições que devem ser verdadeiras imediatamente após a execução de uma operação em uma especificação formal. São declarações de propriedades que devem ser garantidas com o término na execução da rotina (MEYER, 1997).

Os parâmetros da ação são usados na construção das expressões das pré e pós-condições. Como OCL é uma linguagem sem efeitos colaterais, ela não pode modificar os valores dos objetos envolvidos no processo (OMG, 2003). Porém os procedimentos ASSL podem realizar tal tarefa. Dessa forma, associando-se procedimentos ASSL às atividades elas passam a ter o poder de transformar o estado dos objetos. Sendo assim, um processo pode apresentar problemas de não conformidade com as RNs. Tomemos o código OCL a seguir como exemplo para as descrições das falhas:

```
context Process::prepararContrato(umCliente:Cliente)
pre PrepararContrato_pre1:  umCliente.bloqueado = false
```

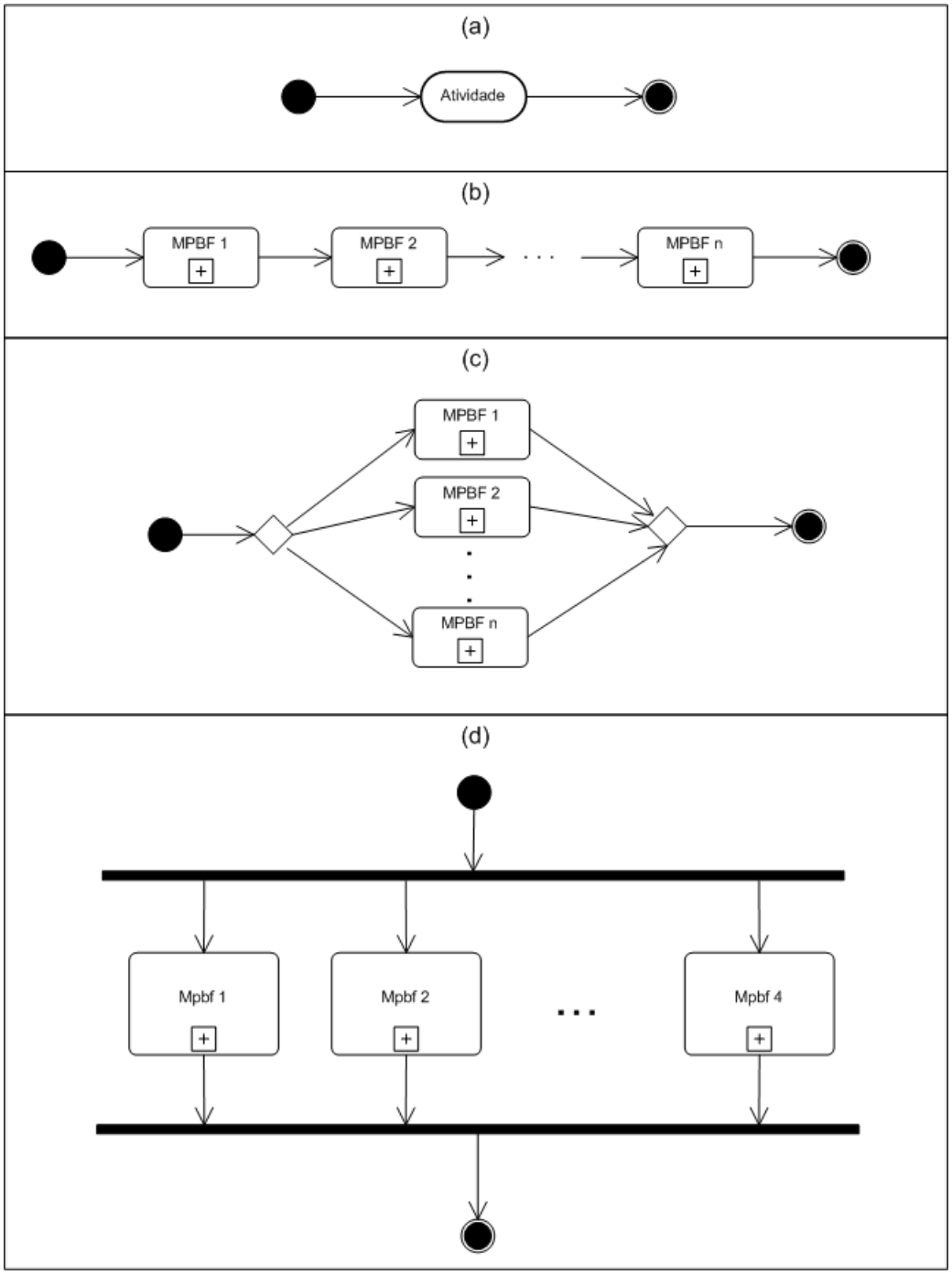


Figura 3.2: Modelos de processos bem formados

```
post PrepararContrato_post02:  self.carro.situacao = 'alugado'

context Carro
inv self.quilometragem < 50000
```

Falha em satisfazer uma pré-condição de atividade: Durante a execução do processo uma atividade pode não ser executada devido à falha em satisfazer uma pré-condição. A figura 3.3 representa um objeto *Cliente* antes da execução da operação preparar Contrato. Seu atributo *bloqueado* tem o valor *true*, o que viola a pré-condição.

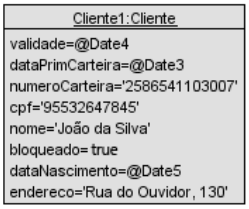


Figura 3.3: Objeto que viola uma pré-condição

Falha em satisfazer uma pós-condição de atividade: Durante a execução do processo a ação de uma atividade pode não ser corretamente encerrada devido à falha em satisfazer uma pós-condição. A figura 3.4 representa um objeto *Carro* após a execução da operação preparar Contrato. Seu atributo *situacao* tem o valor *'selecionado'*, o que viola a pós-condição.

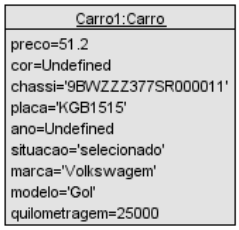


Figura 3.4: Objeto que viola uma pós-condição

Falha em satisfazer uma invariante de classe: Durante a execução do processo a execução de uma ação pode falhar em satisfazer uma invariante de classe, devido às alterações nas instâncias dessa classe. A figura 3.5 representa um objeto *Carro* com o atributo *quilometragem* com o valor *55000*, o que viola a invariante.

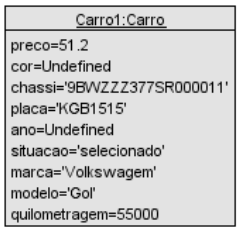


Figura 3.5: Objeto que viola uma invariante

Falha em satisfazer uma decisão: Nos nós de decisão, para cada transição de saída deve haver condições de guarda disjuntas, ou seja, durante a execução do processo, dentre todas as transições de saída, exatamente uma deve ter sua condição de guarda avaliada como verdadeira. Caso nenhuma ou mais de uma transição tenha sua condição de guarda avaliada como verdadeira, o processo não pode decidir por qual transição seguir, então isso também é considerado como uma falha de conformidade do processo com as regras. Dadas as condições de guarda $[\text{carro.preco} < 50]$ e $[\text{carro.preco} = 50]$ combinadas com a figura 3.5 há uma falha, pois neste caso nenhuma condição de guarda satisfeita, já que o atributo *preco* do objeto *Carro1* tem o valor *51.2*.

São esses tipos de problemas que o método proposto tenta encontrar. Durante a execução de um processo, todas as invariantes, pré e pós-condições e condições de guarda envolvidas são avaliadas. Dessa forma, caso alguma das falhas descritas acima acontecer, ela será identificada para que o usuário possa analisá-la e corrigi-la.

3.3 Descrição do Método

Esta seção descreve o método para a validação de conformidade de regras de negócio com processos de negócio, indo desde a especificação das regras e modelos até a simulação. O método será descrito na forma de um algoritmo, para que seu entendimento fique mais fácil.

Construa o modelo de regras de negócio através de um modelo de classes onde todas as regras de negócio estruturais apareçam como invariantes OCL associadas às classes;
Construa o modelo de processo onde as regras aparecem na forma de pré e pós-condições OCL nas atividades, condições de guarda são formadas por expressões booleanas OCL e cada ação das atividades é especificada utilizando semântica de ações;
repita
    Gere um novo cenário de testes sobre o qual a simulação será executada;
    Simule a instância do processo utilizando o algoritmo de simulação aplicado ao cenário previamente gerado;
    se *violação foi encontrada durante a simulação* **então**
        Encerre a simulação;
        Verifique o modelo de processo e as regras de negócio para encontrar onde está a inconsistência;
        Reinicie o algoritmo;
    fim
até *que o nível de confiança de conformidade seja alcançado* ;

Algoritmo 1: Método de verificação de conformidade.

3.3.1 Construir o Modelo de Regras de Negócio

O diagrama de classes define os tipos dos objetos, seus atributos e associações. Cada regra de negócio deve ser especificada na forma de invariantes OCL, usando o modelo de classes de domínio como referência. Os demais tipos de regras devem ser especificados no modelo de processo. As invariantes OCL determinam regras que todas as instâncias da classe a qual essas invariantes estão associadas devem respeitar

a todo momento, a não ser no momento em que uma atividade está em execução. O momento exato da avaliação das invariantes não está definido precisamente na OCL. A figura 3.6 mostra um exemplo de como escrever uma invariante OCL no contexto de uma classe. A invariante *cpfUnico* diz que o atributo cpf da classe *Cliente* deve ser único em todas as instâncias. Já a invariante *maiorIdade* diz que todas as instâncias de *Cliente* devem ter o atributo *idade* com um valor maior ou igual a 18.

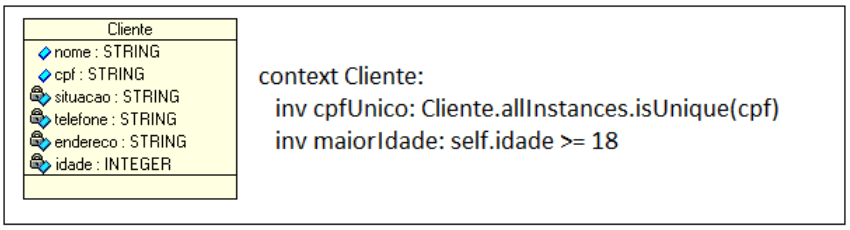


Figura 3.6: Classe UML e suas invariantes em OCL

3.3.2 Construir o Modelo de Processos de Negócio

O modelo de processo compreende um conjunto de atividades. Cada atividade especifica um procedimento que utiliza como entrada objetos e produz saídas, que também são objetos. Esses objetos são especificados pelo diagrama de classes de domínio. Regras de negócio adicionais são inseridas no modelo de processo na forma de pré-condições e pós-condições nas atividades e condições de guarda nas decisões.

A especificação dessas ações é feita através da linguagem ASSL. Para se adequar perfeitamente a este método, adicionamos a cláusula *return* à sintaxe da linguagem ASSL. Essa cláusula é usada para indicar quais foram os objetos de saída de uma atividade. Todos os objetos incluídos na cláusula *return* são armazenados em um objeto padrão *process*, para que possam ser utilizados em atividades posteriores.

A figura 3.7 é um exemplo de como um diagrama de atividades deve ser modelado e anotado com regras OCL e ações ASSL para que o processo seja simulado corretamente. A figura 3.7(a) mostra um trecho de um diagrama de atividades contendo a atividade *Selecionar Carro*, que usa um objeto do tipo *Carro* como insumo e como produto, além de mostrar um nó de decisão, chamado *d2*, que possui uma transição que leva ao fluxo 1 do processo e outra aresta que leva ao fluxo 2. A figura 3.7(b) exibe apenas a classe *Carro*, que é utilizada pela atividade *Selecionar Carro*.

Na figura 3.7(c) estão a pré-condição e pós-condição em OCL e as ações descritas em um procedimento ASSL. Como as pré e pós-condições são escritas em OCL, não se usa colchetes nas expressões. A pré-condição *carroDisponivel* diz que antes que o procedimento seja executado, a propriedade *situacao* do objeto *umCarro* deve ser igual a *'disponivel'*. Já á pós-condição *carroSelecionado* diz que após a execução do procedimento, a propriedade *situacao* do objeto *umCarro* deve ser igual a *'selecionado'*. O procedimento *selecionarCarro* modifica o atributo *situacao* do objeto*umCarro*, dado como entrada, para o valor *'selecionado'* e retorna este objeto, já que no diagrama de classes a atividade *Selecionar Carro* tem este objeto como entrada e saída. O objeto*umCarro* é válido apenas dentro do escopo do procedimento *selecionarCarro* porém, como ele foi retornado por esse procedimento, ele foi armazenado no objeto *process*, visível em todo o escopo do modelo do processo. Para acessar o objeto *umCarro* retornado, basta fazer referência a *process.carro*. Ele fica armazenado com este nome pois é uma instância da classe *Carro*.

Já a figura 3.7(d) exibe as expressões OCL correspondentes às condições de guarda de cada um dos fluxos de saída do nó de decisão d2. Os valores entre colchetes antes da expressão OCL indicam qual fluxo tal expressão habilita, caso seja verdadeira. A expressão [fluxo 1] process.carro.quilometragem <= 50000 diz que a execução do processo deve seguir pelo *fluxo 1* caso o atributo *quilometragem* do objeto *process.carro* (que foi o objeto de saída da atividade *Selecionar Carro*) tenha um valor menor ou

igual a *50000*. A expressão [fluxo 2] process.carro.quilometragem > 50000 diz que a execução do processo deve seguir pelo *fluxo 2* caso o atributo *quilometragem* do objeto *process.carro* tenha um valor maior que *50000*. Note que essas duas condições de guarda são disjuntas, ou seja, as duas nunca serão verdadeiras ao mesmo tempo.

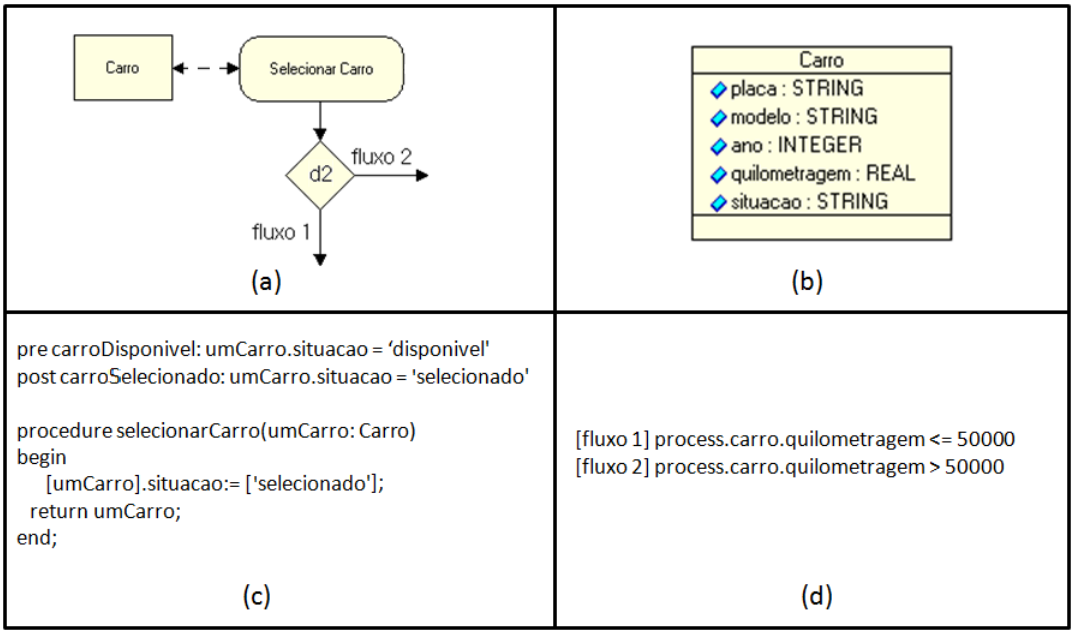


Figura 3.7: Atividade com suas ações e regras e decisão com condições de guarda.

3.3.3 Gerar Casos de Teste

A validação dinâmica de modelos consiste na simulação da execução do modelo a ser validado. Isso exige a geração de um conjunto de casos de testes. No caso deste trabalho, uma instância de processo é descrita pelo diagrama de atividades onde:

1. todos os objetos iniciais, que precisam existir antes do processo ser executado, já estão instanciados;

2. todos os parâmetros de entrada das atividades já receberam um valor especí-

fico;

3. todos os nós de decisão já foram resolvidos

A abordagem original elaborada por Weißleder e Schlingloff (2007a; 2007b) transforma os modelos casados (diagrama de máquina de estados e diagrama de classes) em uma árvore de casos de teste e cria as partições de entradas de testes através da avaliação das expressões OCL. Depois disso, o método gera valores de entrada de testes reais para estas partições.

Porém, adaptando-se o método para este trabalho não será necessário gerar uma árvore de casos de testes, já que, ao invés de diagramas de máquina de estados, são usados diagramas de atividades, que possuem uma estrutura diferente, e que neste trabalho já possuem as condições de guarda e pré e pós-condições diretamente vinculadas a eles. No lugar da árvore de casos de teste geramos o conjunto de caminhos-chave. O tamanho do conjunto de caminhos-chave pode ser facilmente encontrado pelo cálculo da complexidade ciclomática. De acordo com Pressman (2006), a complexidade ciclomática define o número de caminhos independentes – aqui chamados de caminhos-chave – e também fornece um limite superior para a quantidade de testes que devem ser conduzidos para que se garanta que todos os nós e transições do diagrama sejam executados pelo menos uma vez.

No caso do diagrama de atividades, os nós de decisão não são necessariamente binários, ou seja, de um nó de decisão podem sair 2 ou mais transições. Então, neste caso, para que o cálculo do tamanho do conjunto de caminhos-chave seja correto, é necessário levar em conta também a quantidade de transições de saída existentes em cada nó de decisão. Então, baseando-se no cálculo apresentado por Pressman (2006), a fórmula para o cálculo é apresentada na equação (3.1), onde t_i

é a quantidade de transições de saída do i-ésimo nó de decisão do diagrama.

$$1 + \sum (t_i - 1) \tag{3.1}$$

Os *loops* contidos em caminhos-chave devem ser executados mais de uma vez a fim de se obter testes mais completos e confiáveis. Um número máximo para a execução dos *loops* é estabelecido. Nesta situação será gerado um caso de teste onde o *loop* é executado uma vez. Outro caso de teste onde é executado 2 vezes e assim por diante até que seja gerado o caso de teste com um caminho onde o *loop* é executado até o número máximo.

Existem casos em que algum elemento do processo é inalcançável, ou seja, não é possível executar ou chegar a este elemento em uma execução normal do processo. A figura 3.8 ilustra um caso como este. A pós-condição da atividade ativ 1 é a > b. Então, sempre que ativ 1 for executada com sucesso, a condição a > b será verdadeira. Portando, a CG [a > b] da decisão D1 será sempre verdadeira e a CG [a <= b] será sempre falsa. Ou seja, sempre que o processo for executado com sucesso, a atividade ativ 2 será executada e, consequentemente, a atividade ativ 3 e suas transições de entrada e saída não serão executados. Este processo não precisa ter casos de testes para serem executados neste momento, pois uma verificação preliminar mostraria que ele possui elementos inalcançáveis e isto já figura uma falha que deve ser corrigida antes dos testes serem executados.

Em um DA, os elementos contidos entre um nó *fork* e um nó *join* podem ser considerados como subdiagramas ou, neste trabalho, subMPBF. Então, para gerar um caminho que passa por um *fork*, deve-se gerar os caminhos de todos os subMPBFs. No exemplo da figura 3.9, quando a execução do processo atingir o nó *fork*, todos os subMPBFs deverão ser executados simultaneamente. Então, o caminho para a execução deste processo é:

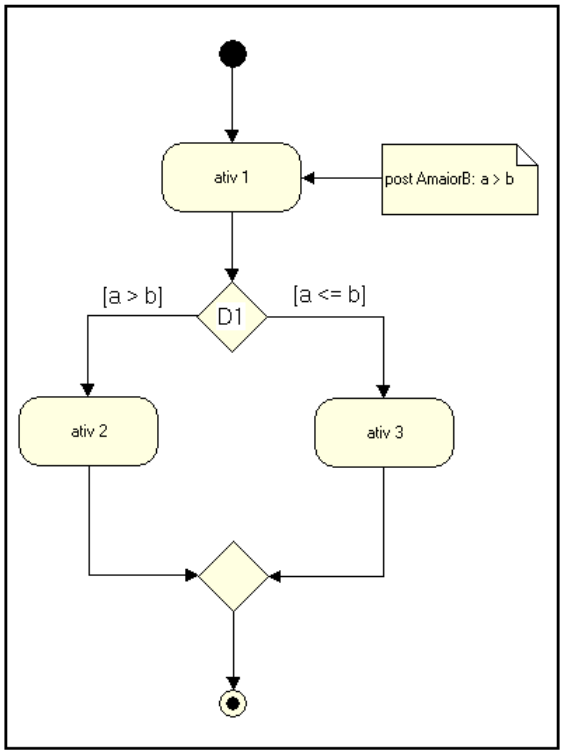


Figura 3.8: Processo com uma atividade inalcançável

$$Início \Rightarrow ativ1 \Rightarrow fork \Rightarrow \left\{ \begin{array}{l} Mpbf1 \\ Mpbf2 \\ Mpbf3 \\ Mpbf4 \end{array} \right\} \Rightarrow join \Rightarrow ativ2 \Rightarrow Fim$$

onde as chaves são usados para representar que todos os subMPBF são executados paralelamente na execução deste caminho.

Cada caminho-chave do conjunto gerado vai corresponder a um ou mais casos de teste. A entrada para os casos de teste são operações parametrizadas na sequência em que aparecem as atividades no caminho-chave. A cada passo da execução do teste, os comportamentos esperados e os atuais são comparados, através da validação das condições, em busca de erros. Cada caso de teste tem que satisfazer a todas as restrições que se encontram ao longo do caminho-chave correspondente.

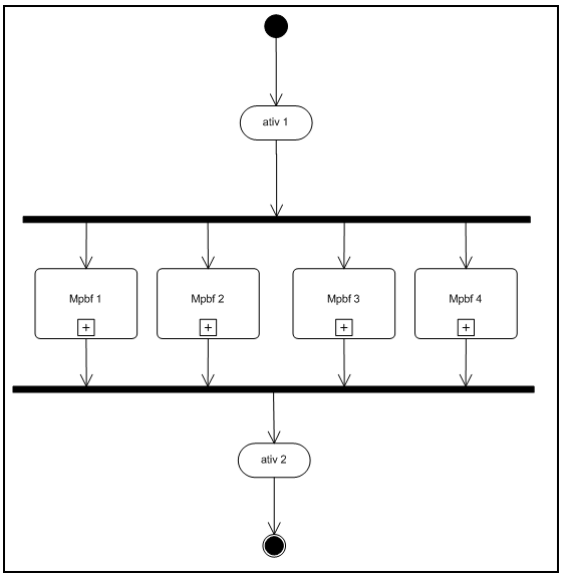


Figura 3.9: Processo com vários fluxos paralelos

Após os caminhos dos casos de teste serem gerados, o próximo passo é gerar os valores de entrada. Os valores de entrada são os valores que os objetos e parâmetros de entrada devem possuir para que a execução do processo percorra o caminho esperado. Utilizando-se os conceitos descritos na seção 2.5 é possível analisar todas as expressões OCL contidas no modelo para gerar partições de intervalos de valores. Nessa partição de valores é possível gerar valores concretos, necessários para que a execução do processo siga o curso esperado, ou seja, os caminhos dos casos de teste previamente gerados. Outra abordagem é gerar valores que estão fora dos intervalos definidos, para verificar se a simulação encontrará erros ao executas tais valores. Para exemplificar a geração dos casos de teste são necessários os modelos e as regras de negócio, por este motivo, um exemplo prático é descrito de forma detalhada na seção 5.

3.3.4 Simular a Execução de uma Instância de Processo

Aqui apresentamos um algoritmo para simular MPBFs. Se durante a simulação ocorrer alguma violação às restrições OCL, então a simulação para imediatamente e tal violação é explicitada. Caso contrário, a simulação continuará até que o caminho sendo percorrido atinja o nó final.

O algoritmo de simulação é descrito a seguir: o procedimento principal é um *loop* que é executado para cada nó no caminho sendo percorrido desde o nó inicial até o nó final. Para cada nó alcançado no caminho é chamado seu algoritmo de execução, que tem como retorno o próximo nó no caminho. O próximo nó é determinado de acordo com as regras e o estado do sistema. Cada tipo de nó tem um algoritmo específico.

O procedimento principal, que é iterado pelo diagrama de atividades é descrito pelo Algoritmo 2. Os nós **início**, **merge** e **join** têm um procedimento igual, apresentado pelo Algoritmo 3. O procedimento de execução das **atividades** está figurado no Algoritmo 4, já o procedimento de execução de **decisões** é explicitado no Algoritmo 5. O procedimento de nós **fork** está descrito no Algoritmo 6.

```
Resultado: caminho percorrido do nó inicial até o nó final ou até ser
           encontrada uma violação
início
    Iniciando pelo nó inicial;
    enquanto caminho percorrido não atingiu o nó final e não houve
    nenhuma violação faça
        execute o procedimento do nó atual, adicione o nó atual ao caminho e
        marque o nó retornado por esse procedimento como o novo nó atual;
    fim
    retorna o caminho percorrido;
fim
```

Algoritmo 2: Procedimento principal de simulação

```
Saída: próximo nó a ser executado
início
| retorna o nó que é vizinho de saída deste nó;
fim
```

Algoritmo 3: Procedimento de execução dos nós Inicial, Merge e Join

```
Saída: próximo nó a ser executado ou mensagem de violação
início
| valide as invariantes e pré-condições OCL desta atividade;
| se não houve violações então
| | execute as ações associadas a esta atividade;
| | valide as invariantes e pós-condições OCL desta atividade;
| | se não houve violações então
| | | retorna nó que é vizinho de saída dessa atividade;
| | fim
| | senão
| | | retorna mensagem de violação de invariantes ou pós-condições
| | | OCL desta atividade;
| | fim
| fim
| senão
| | retorna mensagem de violação de invariantes ou pré-condições OCL
| | desta atividade;
| fim
fim
```

Algoritmo 4: Procedimento de execução do nó Atividade

Saída: próximo nó a ser executado ou mensagem de violação
Seja *contV* um contador de transições verdadeiras;
contV := 0;
para cada *transição de saída deste nó* **faça**
  se *as condições de guarda dessa transição foram avaliadas como verdadeiras* **então**
    marque essa transição como verdadeira;
    incremente o valor de *contV*;
  fim
fim
se *contV = 1* **então**
  retorna *o nó de destino na última transição marcada como verdadeira;*
senão
  retorna *mensagem informando a violação ocorrida neste nó;*
fim

Algoritmo 5: Procedimento de execução do nó Decisão

Saída: próximo nó a ser executado ou mensagem de violação
Seja *subMPBF* um MPBF compreendido entre este nó e o nó Join correspondente;
para cada *subMPBF* **faça**
  inicie a execução deste *subMPBF*;
fim
enquanto *algum subMPBF ainda estiver sendo executado* **faça**
  aguarde o fim da execução de todos os *subMPBF*;
fim
se *houve alguma violação durante a execução dos subMPBF* **então**
  retorna *mensagem informando a violação ocorrida neste nó;*
senão
  retorna *o nó Join correspondente a este nó Fork;*
fim

Algoritmo 6: Procedimento de execução do nó Fork

Estes algoritmos devem ser executados tendo como entrada os diagramas de atividades e de classes de domínio devidamente anotados com as expressões OCL e os procedimentos ASSL. É durante a execução destes algoritmos que todo o processo é simulado, os objetos são gerados e as falhas são encontradas.

4 A FERRAMENTA *PRUV - PROCESS AND RULES VALIDATOR*

O processo de validação de processos e regras de negócio é composto das seguintes etapas:

1. Construir o modelo de regras de negócio;
2. Construir o modelo de processo;
3. Gerar casos de teste;
4. Simular a execução de uma instância do processo.

As duas últimas etapas podem ser particularmente dispendiosas e propensas a erros se realizadas manualmente. Para auxiliar na automatização dessas etapas, desenvolvemos a ferramenta PRUV - Process and RUles Validator, que usa a ferramenta USE como mecanismo de validação das regras em OCL. Ele utiliza como entrada diagramas de atividades e de classes de domínio anotados com as restrições OCL e com as operações em ASSL. Gera automaticamente os caminhos-chave que devem ser percorridos para cobrir 100% do processo. Além disso, em conjunto com a ferramenta USE, automatiza os algoritmos descritos na seção anterior, executando os

procedimentos ASSL e, a cada passo, verifica todas as invariantes e as pré e pós-condições das atividades. Além disso é capaz de decidir dinamicamente os fluxos que devem ser seguidos, através da validação das condições de guarda que devem estar presentes nos nós de decisão.

A ferramenta PRUV também é capaz de indicar quando alguma falha ocorre, dizendo qual regra foi descumprida e o caminho percorrido até o momento da falha. Através da USE é possível visualizar um diagrama de objetos para ver todos os objetos e relacionamentos que a ferramenta PRUV gerou durante a simulação do processo.

4.1 Apresentação

A figura 4.1 exibe a tela principal da ferramenta PRUV, com um projeto UML já aberto. Nessa figura, temos a barra de ferramentas com seus botões numerados. As funções de cada botão são:

1. Abrir projeto UML;
2. Recarregar projeto UML. Se o projeto UML for modificado externamente é necessário recarregá-lo para que as modificações apareçam na PRUV;
3. Fechar projeto;
4. Configurar Simulação;
5. Iniciar Simulação;
6. Iniciar Simulação a partir de um arquivo;
7. Continuar/Pausar Simulação;

8. Parar Simulação;
9. Salvar todas as execuções. Salva todas as ações realizadas na execução de todas as instâncias processo;
10. Salvar a última execução. Salva todas as ações realizadas na última execução do processo.

No menu existem todos os comandos da barra de ferramentas. Além disso há o menu *Skins*, para trocar a aparência da ferramenta e o menu *Help*, com algumas informações sobre a ferramenta.

4.2 Requisitos da Ferramenta

4.2.1 Requisitos do Sistema

Visando uma maior portabilidade, a ferramenta PRUV foi inteiramente desenvolvida na plataforma Java. A linguagem Java, utilizada na plataforma Java é compilada para um *bytecode* [1] que é executado por uma Máquinas Virtuais Java (JVM). Essa característica que faz com que os os programas Java sejam independentes de plataforma, executando em qualquer sistema que possua uma JVM (ORACLE, 2010). Os progras escritos em Java podem ser distribuídos através de *Java Archive* (JAR). JAR é um arquivo compactado usado para distribuir um conjunto de classes Java. É usado para armazenar classes compiladas e metadados associados que podem constituir um programa.

Dessa forma, o único requisito para que a ferramenta seja executada é que o sistema

[1] *Bytecode* é o código compilado para uma forma intermediária de código que é interpretada pelas JVMs. Cada *opcode* tem o tamanho de um *byte* – daí o seu nome.

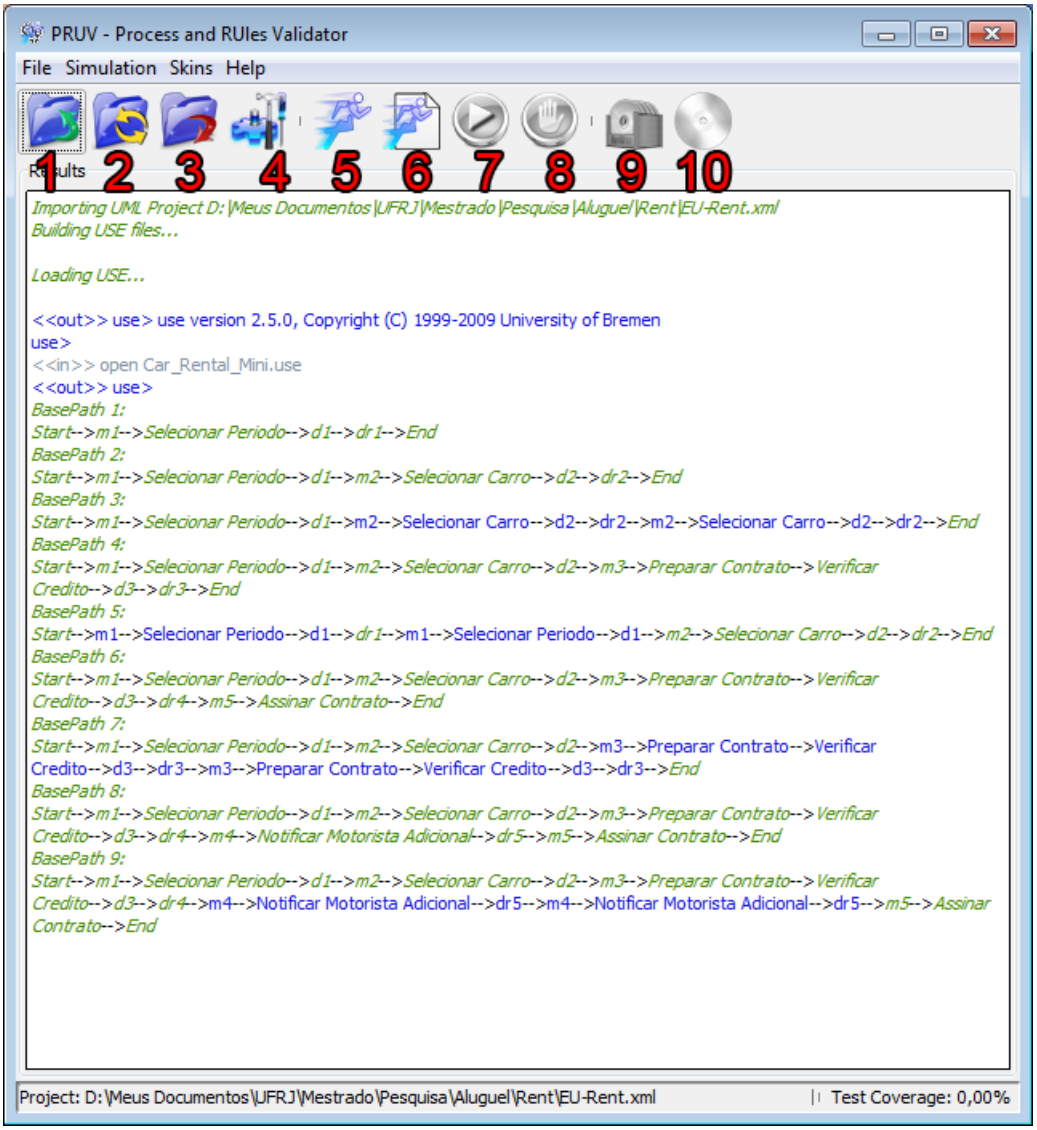


Figura 4.1: Tela Principal da ferramenta PRUV.

operacional possua a JVM instalada. Todo o código da PRUV e suas dependências foram empacotados num único JAR. Portanto, além de ser multiplataforma, a ferramenta PRUV tem uma execução muito simples, já basta executar o arquivo JAR para iniciar a ferramenta, sem que seja necessário nenhum comando ou *script* adicional.

4.2.2 Requisitos para utilizar a ferramenta

A ferramenta PRUV utiliza como entrada diagramas UML anotados com expressões OCL e procedimentos ASSL. Esses diagramas devem ser gerados na ferramenta RAPDIS [2] (MORGADO, 2007). Os diagramas usados são o diagrama de atividades e o digrama de classes. No diagrama de classes, as invariantes devem ser escritas na sua descrição, conforme mostra a figura 4.2.

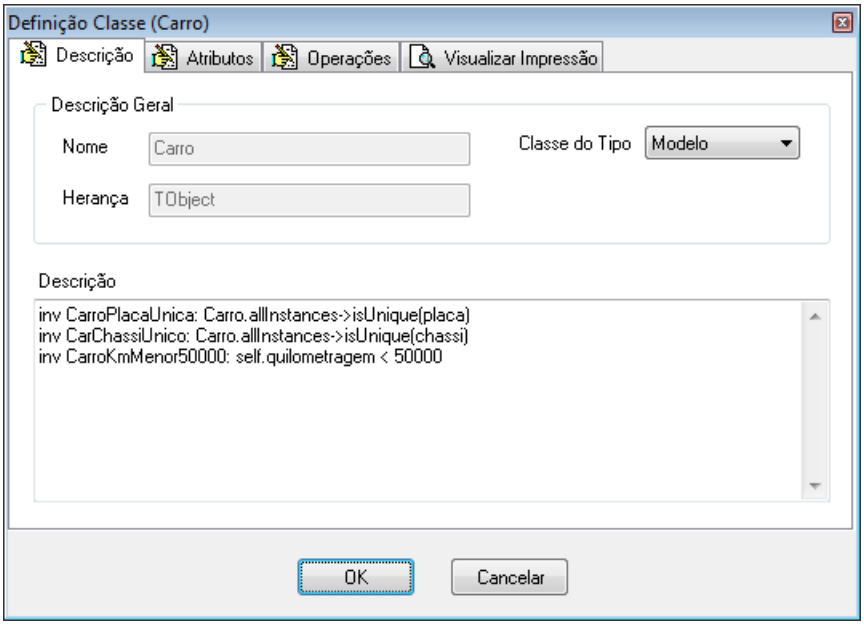


Figura 4.2: Telas de descrição de Classes na RAPDIS.

As anotações das atividades também devem ser feitas na tela de descrição da RAPDIS, conforme mostra a figura 4.3. A descrição das atividades pode conter pré-condições, pós-condições e uma *procedure* ASSL. As atividades, devem incluir em sua *procedure* ASSL uma cláusula de retorno do tipo return <objetoSaida1, objetoSaida2, ...>, onde os objetos que participam da cláusula são os produtos gerados pela atividade.

[2] *Rules And Process for the Development of Information Systems*, pode ser encontrada no endereço http://geti.dcc.ufrj.br/projetos/rapdis/

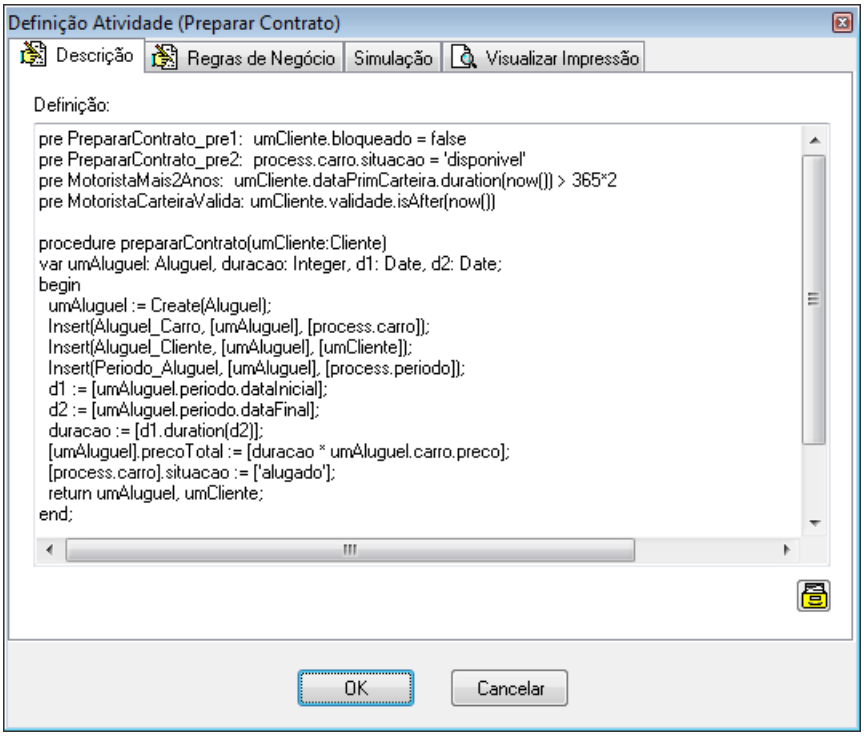


Figura 4.3: Telas de descrição de Atividades na RAPDIS.

Existe também a possibilidade de definir para cada atividade um tempo mínimo, um tempo mais provável e um tempo máximo que sua execução leva dentro da RAPDIS, de acordo com a parte destacada da figura 4.4. Não há uma unidade de tempo especificada, então o usuário pode definir a unidade de tempo de sua preferância A ferramenta PRUV usa esses tempos durante a simulação para atribui um valor específico à execução da respectiva atividade. Esse valor é sorteado através de uma trisbiuição triangular.

Os nós *decisão*, *merge*, *fork* e *join* na RAPDIS não possuem nomes, portanto devem conter em sua descrição uma linha indicando seu nome, para que a ferramenta PRUV possa identificá-los. Exemplo: name = Decisao1. As decisões, além do nome, devem indicar suas condições de guarda, caso sejam decisões computáveis ou indicar que são decisões aleatórias. Para decisões computáveis, as condições de guarda devem

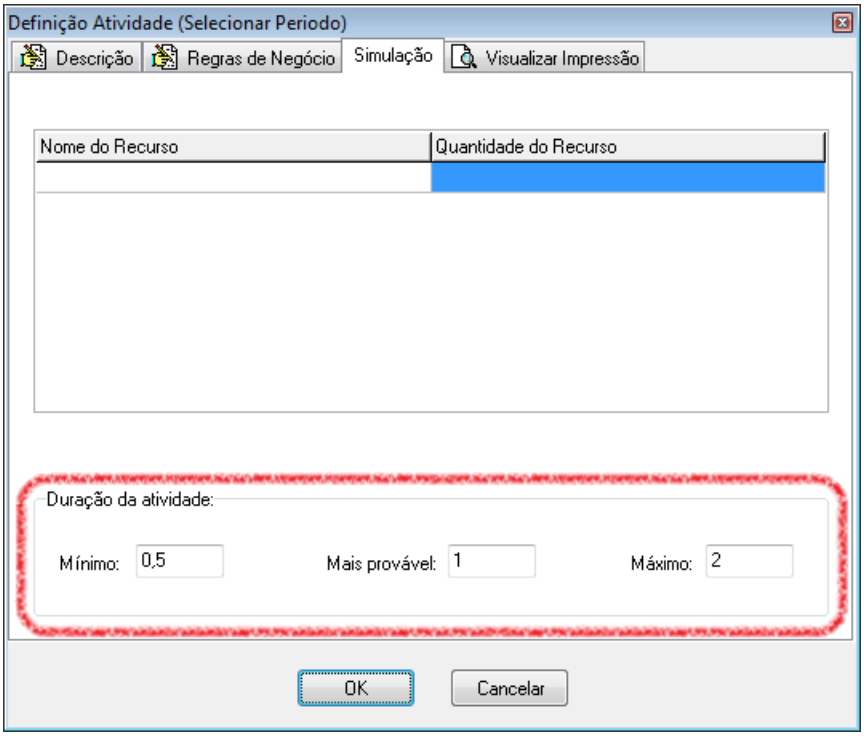


Figura 4.4: Telas de definição de tempos mínimo, mais provável e máximo de uma atividade na RAPDIS.

ser escritas no seguinte formato: [<*nó de destino*>] <*expressão OCL*>. Para indicar que são decisões aleatórias, basta adicionar uma linha com o texto random = true.

A ferramenta ainda utiliza um arquivo contendo procedimentos em ASSL para gerar seus casos de teste. Ela não gera automaticamente os valores dos parâmetros de entrada para a execução das atividades, então este arquivo serve para gerar tais valores. O arquivo deve ter o nome *parameters.assl* e deve estar na pasta raiz do projeto RAPDIS que será usado. Os procedimentos devem retornar os valores que serão as entradas das atividades. Por exemplo, a atividade *Selecionar Periodo* tem em sua descrição o seguinte procedimento em ASSL:

```
procedure selecionarPeriodo(dataInicio:Date,dataFim:Date)
```

```
var periodo: Periodo;
begin
  periodo:= Create(Periodo);
  [periodo].dataInicial:= [dataInicio];
  [periodo].dataFinal:= [dataFim];
  return periodo;
end;
```

Então, o procedimento que irá gerar os valores de entrada para esta atividade deve conter o mesmo nome do procedimento da atividade, porém sem nenhum parâmetro de entrada. Além disso, ele deve retornar os objetos que são os parâmetros de entrada para o procedimento da atividade com os mesmos nomes que estão no procedimento da atividade. Exemplo:

```
procedure selecionarPeriodo()
var dataInicio:Date, dataFim: Date;
begin
  dataInicio := Create(Date);
  [dataInicio].day := [6];
  [dataInicio].month := [6];
  [dataInicio].year := [2010];
  dataFim := Create(Date);
  [dataFim].day := [14];
  [dataFim].month := [7];
  [dataFim].year := [2010];
  return dataInicio, dataFim;
end;
```

A cláusula *return* dos procedimentos nas descrições das atividades serve para armazenar os objetos de saída das atividades. Esse recurso foi criado para a ferramenta PRUV, já que a linguagem ASSL não suporta retorno de procedimentos. Todos os retornos são armazenados no objeto *process*. Este objeto é global, visível em todo o escopo do processo e é um objeto embutido na ferramenta. Portanto, não é possível criar outros objetos ou classes com este nome. É neste objeto que ficam todas as operações das atividades. Então, o procedimento *selecionarPeriodo* do exemplo acima está no escopo do objeto *process*, assim como todos os outros procedimentos escritos no diagrama de atividades.

4.3 Utilizando da Ferramenta

4.3.1 Abrindo um Projeto

Para começar a utilizar a ferramenta, o projeto RAPDIS que contém as especificações descritas na seção anterior deve ser aberto. O arquivo a ser aberto é o XML contido na pasta raiz do projeto RAPDIS. Ao abrir um projeto, a PRUV irá exibir uma tela para a seleção dos diagramas a serem usados, conforme mostra a figura 4.5. O usuário deve escolher entre um diagrama de atividades e um diagrama de classes, pois a ferramenta RAPDIS permite a criação de vários deles em um mesmo projeto.

Caso o diagrama de classes escolhido possua alguma classe modelada com os nomes *Date*, *Time* ou *DateTime* a ferramenta PRUV irá perguntar se o usuário deseja usar as classes que ele modelou ou se deseja usar as classes que já fazem parte da ferramenta. A vantagem de usar as classes já embutidas na PRUV é que o usuário não precisa definir nenhum atributo ou operação para essas classes, basta apenas declará-las no modelo, que a ferramenta irá incluir todos os atributos, operações

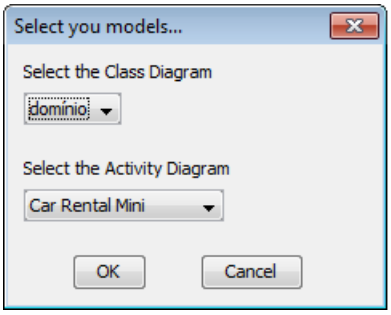


Figura 4.5: Telas de seleção dos diagramas RAPDIS a serem usados.

e invariantes pertinentes a essas classes. A figura 4.6 ilustra a pergunta feita ao usuário pela ferramenta.

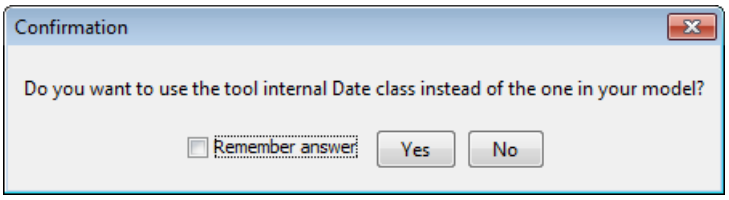


Figura 4.6: Telas para confirmar se as classes internas serão usadas.

Após isso o projeto é carregado, os arquivos RAPDIS são importados para a ferramenta e a ferramenta USE, usada para realizar as validações OCL é carregada para se comunicar com a PRUV. Após abrir o projeto, a ferramenta PRUV gera automaticamente todos os caminhos-chave que poderão ser usados nos casos de teste A figura 4.1 mostra os caminhos-chave gerados. Quando há *loops* no caminho chave, a ferramenta os identifica destacando-os com uma letra azul, enquanto os nós que não fazem parte de *loops* são exibidos na cor verde.

4.3.2 Configurando a Simulação

Para configurar a simulação, basta clicar no botão 4 da figura 4.1 ou acessar essa funcionalidade pelo menu. A tela mostrada na figura 4.7 será exibida. A opção 1

é para configurar quantas vezes um cenário deve ser gerado. A opção 2 é configura quantos segundos a simulação é pausada entre um nó e outro. Essa opção serve para o usuário que deseja ver a simulação sendo executada passo a passo. A opção 3 é para configurar se os objetos do processo gerados nas execuções anteriores devem ser apagados e novos objetos devem ser gerados e a opção 4 é para que os objetos gerados nas execuções anteriores continuem existindo nas execuções seguintes. A opção 5 configura as decisões aleatórias para sortear automaticamente qual o fluxo a simulação deve seguir e a opção 6 configura as decisões aleatórias para perguntarem qual será o fluxo a ser executado. Caso o arquivo *parameters.assl* contiver procedimentos que retornem valores para essas decisões, então a PRUV irá decidir qual será o fluxo a partir desse arquivo, sem a necessidade da ação do usuário.

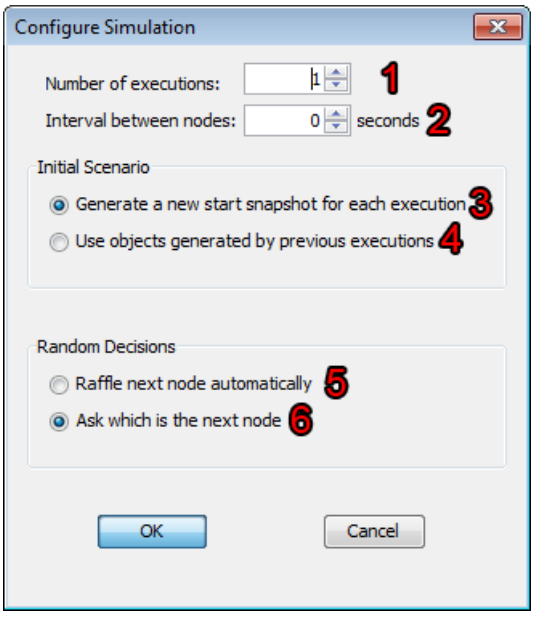


Figura 4.7: Tela de configuração da simulação.

4.3.3 Simulando o Processo

Para iniciar a simulação, basta clicar no botão 5 da figura 4.1 ou acessar essa funcionalidade pelo menu. Ao iniciar a simulação a ferramenta irá automaticamente gerar todos os objetos iniciais necessários para que a simulação comece. Conforme a simulação for executada ela irá validar todas as pré-condições, pós-condições e invariantes através da ferramenta USE. Para executar uma atividade, a PRUV irá gerar os valores para os parâmetros de entrada através do arquivo *parameters.assl*. Caso esse arquivo não exista ou não houver um procedimento para gerar os parâmetros de entrada para a atividade, então uma tela será aberta para que o usuário entre com os valores manualmente, conforme mostra a figura 4.8. A cada só executado, todos os passos da execução são mostrados. Por exemplo:

```
Node: Selecionar Periodo
<<in>> !openter process selecionarPeriodo(Date1,Date2)
<<out>> Date1,Date2
use>
<<in>> gen start -s -b Car_Rental_Mini.assl
selecionarPeriodo(process,Date1,Date2)
<<out>> !create Periodo1 : Periodo
!set Periodo1.dataInicial := @Date1
!set Periodo1.dataFinal := @Date2
!set process.periodo := @Periodo1
check state (1): valid state.
<<out>> Generated result (system state) accepted.
<<out>> postcondition ‘SelecionarPeriodo1’ is true
postcondition ‘SelecionarPeriodo2’ is true
postcondition ‘SelecionarPeriodo3’ is true
use>
```

```
Total time: 1.2123980183819092
```

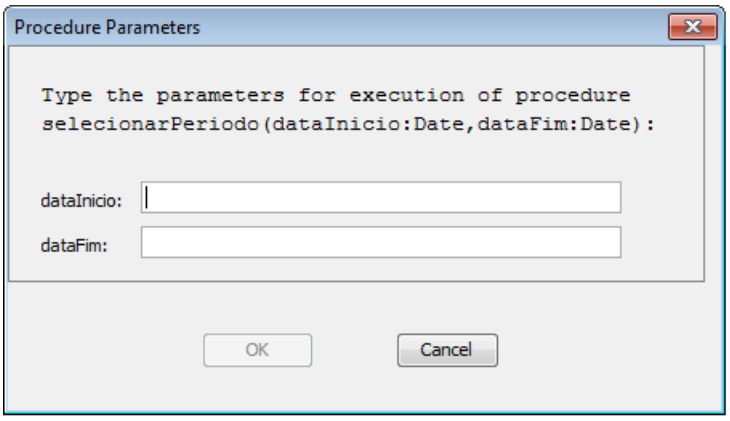


Figura 4.8: Tela de entrada manual para os parâmetros dos procedimentos.

Ao início da execução das atividades a validação das pré-condições é mostrada. Se todas forem verdadeiras a atividade é executada. Ao final da execução da atividade, a validação das pós-condições é mostrada. Caso todas sejam verdadeiras a simulação continua. Além disso é mostrado o tempo total que o processo gastou para ser executado. Esse tempo é a soma dos tempos que foram atribuídos a cada atividade pela distribuição triangular, usando como parâmetros os tempos mínimo, mais provável e máximo, conforme foi explicado na seção 4.2.2.

Para a execução das decisões, a ferramenta avalia as condições de guarda para saber qual por qual fluxo seguir a execução. Após avaliar as condições de guarda, a ferramenta segue pelo fluxo onde a condição de guarda foi avaliada como verdadeira. No exemplo abaixo, a condição de guarda que foi avaliada como verdadeira levava ao nó *m2*, portanto esse nó foi o nó por onde a simulação prosseguiu:

```
Node: d1
<<in>> ? process.periodo.dataInicial.duration(
  process.periodo.dataFinal) <= 90
```

```
<<out>> -> true : Boolean
use>
<<in>> ? process.periodo.dataInicial.duration(
  process.periodo.dataFinal) > 90
<<out>> -> false : Boolean
use>
Node: m2
```

É possível pausar/continuar ou parar a simulação (botões 7 e 8 da figura 4.1, respectivamente). Além disso, informações extras são exibidas na barra de *status* localizada na parte inferior da ferramenta. A figura 4.9 mostra um caso em que o nó sendo executado é o *dr5*, a simulação está na primeira execução do processo e os testes já alcançaram a cobertura de 46,15% do diagrama de atividades.

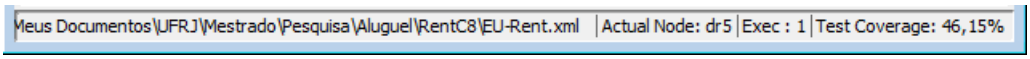


Figura 4.9: Barra de *status* da ferramenta.

Caso alguma violação seja encontrada, a simulação para e a mensagem de erro exibindo a violação é exibida. Veja o exemplo abaixo:

```
Total time up to this node: 7.061160748151344
Path to reach this node: Start --> m1 --> Selecionar Periodo --> d1
  --> m2 --> Selecionar Carro --> d2 --> m3 --> Preparar Contrato
ERROR AT EXECUTION 1
Error found is: PRE-CONDITION FALSE: precondition ‘PrepararContrato_
  pre2’ is false
```

Neste caso, o usuário deve analisar a mensagem para entender qual foi o erro e porque ele aconteceu. Após a análise e correção do erro, o projeto deve ser recarregado

(botão 2 da figura 4.1) para que as simulações sejam realizadas novamente.

Caso a simulação consiga executar todo o processo sem erros, a seguinte mensagem é exibida:

```
Node: End
<<out>> use>
Node Start is visited.
Node Selecionar Carro is visited.
...
Node Assinar Contrato is not visited.
...
Link Selecionar Carro-d2 is visited.
Link m2-Selecionar Carro is visited.
Link dr4-m4 is not visited.
...
Total time of process execution: 25.363580672076964
Path to reach the end of process: Start --> m1 --> Selecionar Periodo
  --> d1 --> m2 --> Selecionar Carro --> d2 --> m3 --> Preparar
  Contrato --> Verificar Credito --> d3 --> dr3 --> End
<<EXECUTION FINISHED>>
```

4.3.4 Exportando e Importando Simulações

É possível exportar todos os passos que a ferramenta executou durante a simulação do processo para arquivos de texto. Pode-se salvar todas as execuções do processo ou apenas a última execução (botões 9 e 10 da figura 4.1, respectivamente). Esses arquivos são úteis pois eles podem ser importados pela própria ferramenta PRUV ou

pela ferramenta USE. Ao importar do RAPDIS um projeto, a PRUV gera automaticamente um arquivo com o formato usado pela USE. Dessa forma, todo o modelo pode ser aberto nessa ferramenta. Após abrir o modelo, é possível executar passo a passo, dentro da USE a simulação. A USE gera diagramas de objetos e de sequência, por isso pode ser útil para a depuração de erros. O arquivo de simulação gerado também pode ser importado pela própria ferramenta PRUV (botão 6 da figura 4.1). Essa funcionalidade existe para que o usuário possa reproduzir exatamente a mesma simulação posteriormente caso seja necessário.

5 EXEMPLO DE APLICAÇÃO

Para demonstrar como o método é realizado passo a passo utilizaremos um exemplo. Trata-se de uma locadora de carros, chamada EU-Rent. A descrição detalhada da locadora e de suas regras de negócio encontra-se documentada pela OMG (2008). Focaremos no processo de aluguel imediato, sem reservas. A modelagem do diagrama de classes e do diagrama de atividades, incluindo as anotações em OCL para este exemplo foi realizada utilizando a ferramenta case RAPDIS. A simulação do processo foi realizada utilizando-se a ferramenta PRUV.

Este exemplo tem o objetivo de mostrar como o método pode ser usado a fim de encontrar inconsistências entre os processos de negócio e as regras de negócio. Também queremos mostrar que o método é capaz de auxiliar os usuários a identificarem alterações que trazem melhorias para o processo sendo analisado.

Abaixo listamos uma pequena parte das regras que compõem este exemplo:

Regra 1 O tempo máximo de duração de um aluguel é de 90 dias.

Regra 2 A validade do cartão de crédito deve ser maior que a data final da locação.

Regra 3 O número de CPF e de habilitação do cliente e dos motoristas é único.

Regra 4 Se uma locação tem algum motorista adicional, então a tarifa deve ser acrescida de 25%.

Regra 5 Uma locação deve ter no máximo dois motoristas adicionais cadastrados.

Regra 6 Uma locação não pode usar um carro com mais de 50.000 km rodados.

Regra 7 Um cliente bloqueado não pode alugar um carro.

Regra 8 Os carros podem ser selecionados apenas se estiverem disponíveis.

Regra 9 O saldo do cartão de crédito tem que ser maior ou igual a tarifa do aluguel.

Regra 10 Clientes e motoristas devem ser habilitados para dirigir há pelo menos 2 anos.

Regra 11 Quando o contrato for preparado o carro deve ser reservado.

Regra 12 A placa e o chassi de um carro devem ser únicos.

Regra 13 Um aluguel sempre deve ter um período especificado.

Regra 14 A habilitação do cliente e dos motoristas devem estar na validade.

Regra 15 A data final do período escolhido deve ser maior que a data inicial.

Regra 16 A data de início do aluguel deve ser considerada como a data atual.

Regra 17 O aluguel é confirmado após a assinatura do contrato.

Regra 18 O preço total do aluguel é calculado multiplicando-se o preço do carro pela quantidade de dias do período escolhido.

As seções seguintes explicam todos os passos do método sendo realizados com este exemplo. Todas as regras numeradas nessas seções se referem às regras listadas acima.

5.1 Criar Modelos

Para começar, devemos criar o diagrama de classes de domínio UML anotado com invariantes OCL. Analisando-se a descrição do negócio com suas regras, disponível no documento da OMG, chegamos ao modelo de regras mostrado na figura 5.1. Algumas classes ficaram sem invariantes OCL e as regras relacionadas a essas classes serão escritas na forma de pré e pós-condições, pois são regras que dependem do contexto da execução do processo.

Na implementação da OCL na ferramenta USE não existe a possibilidade de modelar classes ou operações estáticas, nem tampouco uma classe já definida para lidar com datas ou horas. Por isso, foram criadas classes do tipo Date, Time e DateTime que ficam embutidas na ferramenta PRUV. Portanto, basta incluir uma dessas classes no diagrama de classes para que todas as operações já definidas na classe embutida na ferramenta como *now*, *isAfter*, *isBefore*, entre outras, possam ser usadas no modelo. Como estas classes não podem ser estáticas, esses métodos sempre devem ser chamados a partir de um objeto já instanciado dessas classes. Dentro do contexto das atividades, o objeto *process* pode ser usado. É nele que estão implementadas todas as operações das atividades do diagrama. Ele herda da classe *Date* e é também um objeto embutido na ferramenta, responsável por armazenar todos os objetos de saída das atividades. Dentro do contexto de *process*, a palavra *process* pode ser omitida ou substituída por *self*.

As invariantes das classe *Carro* refletem a **Regra 12** as da classe *Motorista* refletem

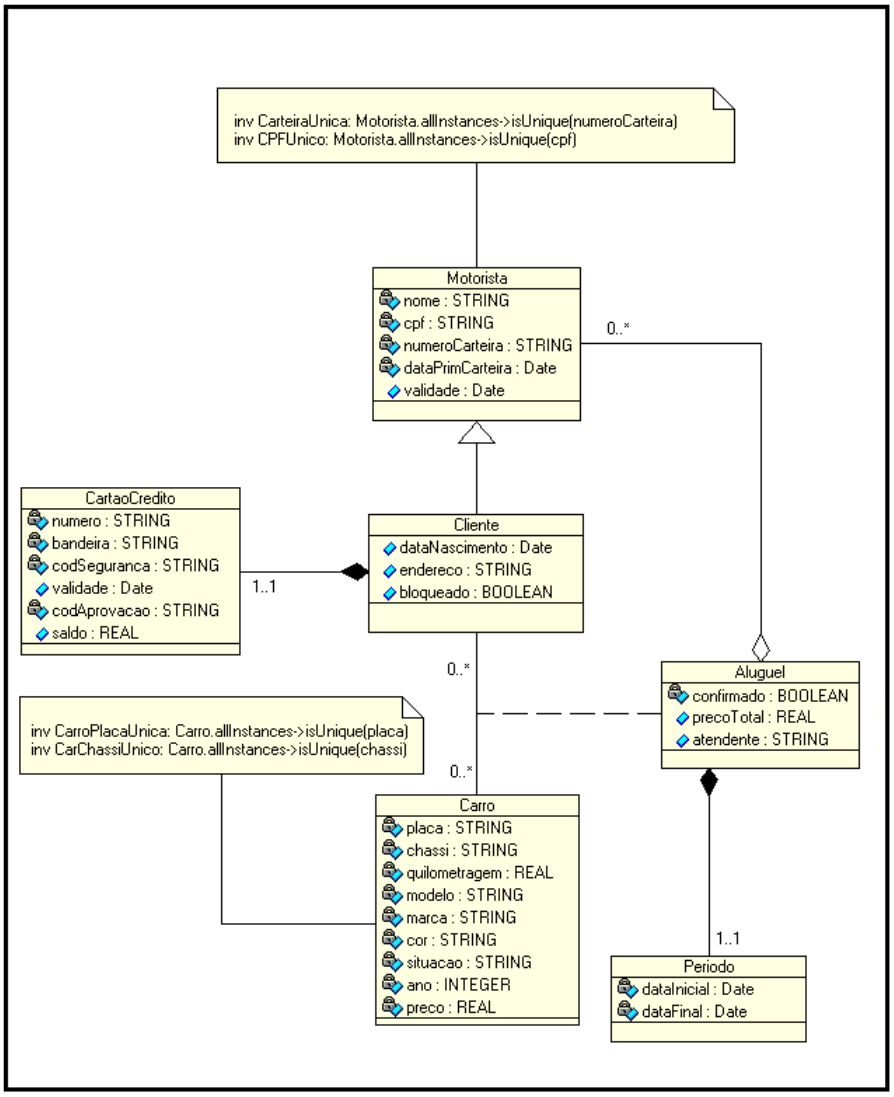


Figura 5.1: Diagrama de classes de domínio anotado com OCL

a **Regra 3**

O segundo passo é construir o modelo de processo. Neste caso, o processo de negócio em questão é o aluguel imediato. Este modelo é composto por um diagrama de atividades UML, onde as atividades são anotadas com regras OCL e possuem procedimentos em ASSL para manipulação do cenário de objetos, e as decisões contêm

Tabela 5.1: Especificação das siglas da figura 5.2

Sigla	**Significado**
m1	merge 1
d1	decisão 1
[1.1]	condição de guarda d1-dr1
[1.2]	condição de guarda d1-m2
dr1	decisão aleatória 1
m1	merge 1
m2	merge 2
d2	decisão 2
[2.1]	condição de guarda d2-dr2
[2.2]	condição de guarda d2-Preparar Contrato
dr1	decisão aleatória 2
m3	merge 3
d3	decisão 3
[3.1]	condição de guarda d3-dr4
[3.2]	condição de guarda d3-dr3
dr3	decisão aleatória 3
dr4	decisão aleatória 4
m4	merge 4
m5	merge 5
dr5	decisão aleatória 5

condições de guarda escritas através de expressões OCL. Analisando a descrição do processo e das regras do negócio, podemos chegar ao modelo da figura 5.2. Para facilitar a visualização, criamos a tabela 5.1 para especificar o significado das siglas presentes no diagrama e separamos as regras deste diagrama e as listamos a seguir.

Selecionar Periodo

```
pre SelecionarPeriodo_pre: periodo.dataFinal.isAfter(
  periodo.dataInicial)
```

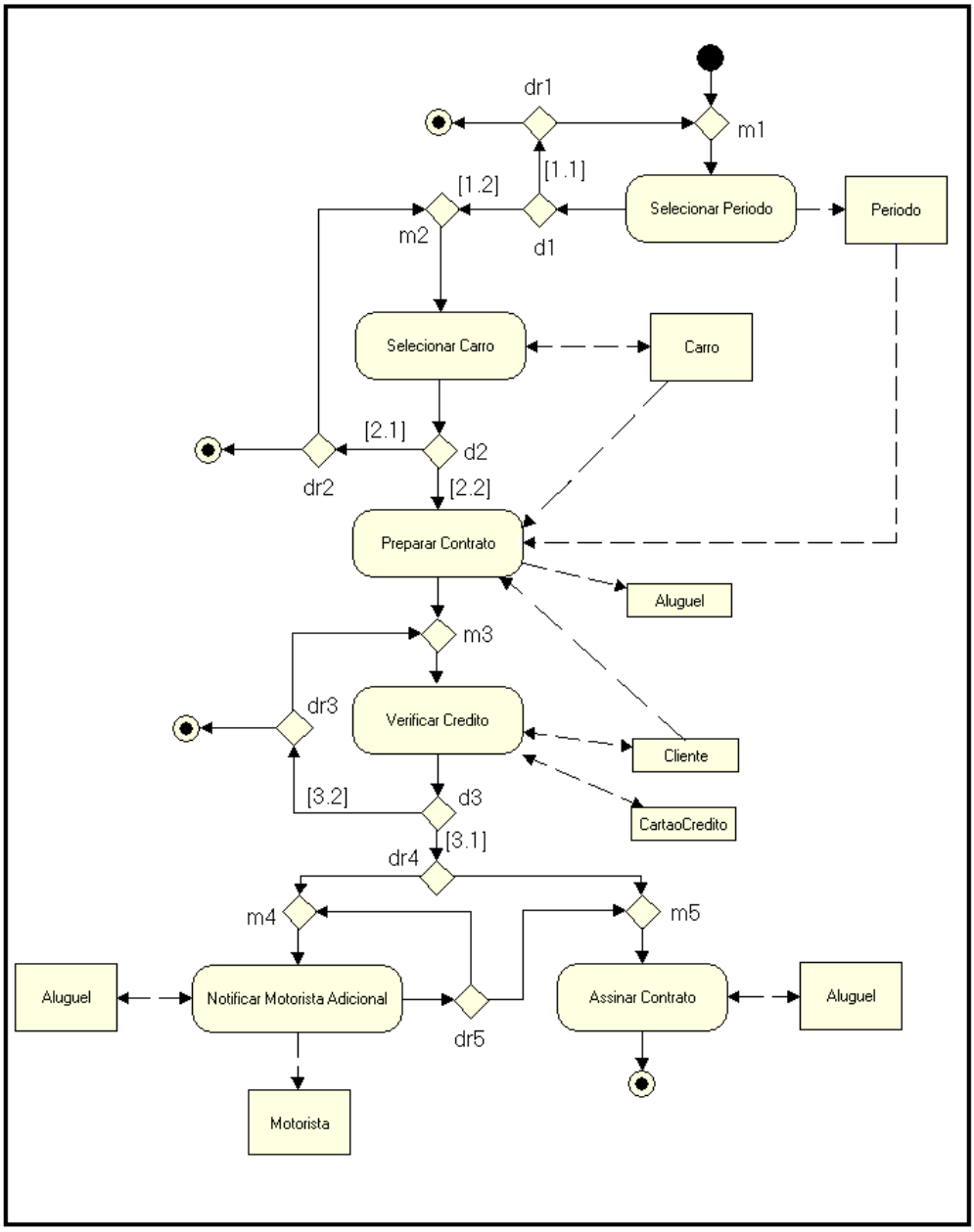


Figura 5.2: Diagrama de atividades do processo aluguel imediato

```
procedure selecionarPeriodo(dataInicio:Date,dataFim:Date)
var periodo: Periodo;
```

```
begin
  periodo:= Create(Periodo);
  [periodo].dataInicial:= [dataInicio];
  [periodo].dataFinal:= [dataFim];
  return periodo;
end;

post SelecionarPeriodo1: periodo.dataInicial = dataInicio
post SelecionarPeriodo2: periodo.dataFinal = dataFim
post SelecionarPeriodo3: periodo.dataInicial.isEqual(now())
```

A pré-condição *SelecionarPeriodo_pre* expressa a **Regra 15** O procedimento *selecionarPeriodo* cria um novo periodo, atribui o valor de *dataInicio* como sua *dataInicial* e *dataFim* como sua *dataFinal* e retorna esse periodo, que ficará armazenado como *process.periodo*. A pós-condição *SelecionarPeriodo3* diz que a *dataInicial* expressa a **Regra 16**.

Selecionar Carro

```
pre SelecionarCarro_pre1: umCarro.situacao = 'disponivel'

procedure selecionarCarro(umCarro: Carro)
begin
  [umCarro].situacao := ['selecionado'];
  return umCarro;
end;

post SelecionarCarro_post01: umCarro.situacao = 'selecionado'
```

A pré-condição *SelecionarCarro_pre1* exprime a **Regra 8**. O procedimento *sele-*

cionarCarro muda o valor de *umCarro.situacao* para *'selecionado'* e retorna esse objeto *umCarro*, que ficará armazenado como *process.periodo*. A pós-condição *SelecionarCarro_post01* diz que o carro que foi usado como entrada nessa atividade deve ter seu atributo *situacao* modificado para *'selecionado'*.

Preparar Contrato

```
pre PrepararContrato_pre1: umCliente.bloqueado = false
pre PrepararContrato_pre2: process.carro.situacao = 'disponivel'
pre DirigeMais2Anos:  umCliente.dataPrimCarteira.duration(now())
  >= 365*2
pre ClienteCarteiraValida: umCliente.validade.isAfter(now())

procedure prepararContrato(umCliente:Cliente)
var umAluguel: Aluguel;
begin
  umAluguel := Create(Aluguel);
  Insert(Aluguel_Carro, [umAluguel], [process.carro]);
  Insert(Aluguel_Cliente, [umAluguel], [umCliente]);
  Insert(Periodo_Aluguel, [umAluguel], [process.periodo]);
  [umAluguel].precoTotal := [process.periodo.dataInicial.duration(
  process.aluguel.periodo.dataFinal) * process.aluguel.carro.preco]
  [process.carro].situacao := ['alugado'];
  return umAluguel, umCliente;
end;

post PrepararContrato_post01: (umAluguel.cliente = umCliente and
  umAluguel.carro = process.carro and umAluguel.periodo = process.periodo
post PrepararContrato_post02: process.carro.situacao = 'alugado'
post PrepararContrato_post03: umAluguel.precoTotal =
```

```
process.periodo.dataInicial.duration(process.aluguel.periodo.dataFinal)
* process.aluguel.carro.preco
```

A pré-condição *PreparарContrato_pre1* expressa a **Regra 7**. A pré-condição *PreparарContrato_pre2* diz que *process.carro.situacao* (*process.carro* é o objeto retornado na atividade Selecionar Carro) deve ter o valor *'selecionado'*. A pré-condição *DirigeMais2Anos* reflete a **Regra 10** e a pré-condição *ClienteCarteiraValida* reflete a **Regra 14**. O procedimento *prepararContrato* instancia um objeto do tipo *Aluguel* com o nome *umAluguel* e cria o relacionamento entre ele e os objetos *process.carro*, *umCliente*, *process.periodo*. Depois coloca no atributo *precoTotal* o valor calculado como a duração do aluguel multiplicado pelo preço de uma diária do carro, muda o atributo *process.carro.situacao* para *'alugado'* e retorna os objetos *umAluguel* e *umCliente* para *process*. A pós-condição *PreparарContrato_post01* verifica se os objetos *umCliente*, *process.carro* e *process.period* foram devidamente associados ao objeto *umAluguel*. Já a pós-condição *PreparарContrato_post02* verifica se o atributo *process.carro.situacao* recebeu o valor *'alugado'* e a pós-condição *PreparарContrato_post03* verifica se o preço total do aluguel foi calculado corretamente, para respeitar a **Regra 18**.

Verificar Credito

```
pre CartaoValido: umCartao.validade.isAfter(process.periodo.dataFinal)

procedure verificarCredito(umCartao: CartaoCredito)
begin
  [umCartao].codAprovacao := [‘aprovado’];
  Insert(CartaoCredito_Cliente, [process.cliente], [umCartao]);
  return umCartao;
end;
```

```
post VerificarCredito_post1: process.cliente.cartaoCredito = umCartao
post VerificarCredito_post2: process.cliente.cartaoCredito.
  codAprovacao.isDefined
post VerificarCredito_post3: process.cliente.cartaoCredito = umCartao
```

A pré-condição *PreparарContrato_pre1* expressa a **Regra 2**. O procedimento *verificarCredito* modifica o valor do atributo *umCartao.codAprovacao* para *'aprovado'* e cria o relacionamento entre *process.cliente* e *umCartao*. Depois retorna o objeto *umCartao*. As pós-condições verificam se o procedimento *verificarCredito* realizou as alterações corretamente.

Notificar Motorista Adicional

```
pre MotoristaMais2Anos:  umMotorista.dataPrimCarteira.duration(now())
  > 365*2
pre MotoristaCarteiraValida: umMotorista.validade.isAfter(now())
pre Max2Adicionais: process.aluguel.motoristaAdicional->size() < 2

procedure notificarMotoristaAdicional(umMotorista: Motorista)
begin
  Insert(Motorista_Aluguel, [process.aluguel], [umMotorista]);
  if process.aluguel.motoristaAdicional->size() = 1
  begin
    [process.aluguel].precoTotal := [process.aluguel.precoTotal* 1.25];
  end;
  return umMotorista;
end;

post NotificarMotoristaAdicional_post01: process.aluguel.precoTotal =
  process.aluguel.motoristaAdicional->size() = 1 implies
```

```
  process.aluguel.precoTotal = self.aluguel.precoTotal@pre* 1.25
post NotificarMotoristaAdicional_post02:
  process.aluguel.motoristaAdicional->size() =
  process.aluguel.motoristaAdicional@pre->size() +1
```

A pré-condição *MotoristaMais2Anos* reflete a **Regra 10**. A pré-condição *MotoristaCarteiraValida* expressa a **Regra 14**. A pré-condição *Max2Adicionais* verifica se o número de motoristas adicionais do aluguel é menor que dois, para que a **Regra 5** não seja violada. O procedimento *notificarMotoristaAdicional* adiciona um novo motorista ao aluguel e verifica se a quantidade de motoristas adicionais é um para adicionar a multa de 25% ao valor do aluguel. Depois retorna o objeto *umMotorista*. A pós-condição *NotificarMotoristaAdicional_post01* reflete a **Regra 4**. Já a pós-condição *NotificarMotoristaAdicional_post02* verifica se o motorista foi realmente adicionado ao aluguel, comparando se a quantidade de motoristas após a execução da atividade é igual à quantidade de motoristas antes da execução da atividade mais um.

Assinar Contrato

```
pre assinarContrato_pre01: process.cartaoCredito.saldo >=
  process.aluguel.precoTotal

procedure assinarContrato(umAtendente: String)
begin
  [process.aluguel].atendente := [umAtendente];
  [process.aluguel].confirmado := [true];
end;
post assinarContrato_post01: process.aluguel.confirmado = true
```

A pré-condição *assinarContrato_pre01* reflete a **Regra 9**. O procedimento *assinar-*

Contrato modifica o atributo *process.aluguel.atendente* para o valor *umAtendente* e modifica o atributo *process.aluguel.confirmado* para *true*. A pós-condição *assinarContrato_post01* expressa a **Regra 17**.

Decisão d1

```
[dr1] process.periodo.dataInicial.duration(process.periodo.dataFinal)
  > 90
[m2] process.periodo.dataInicial.duration(process.periodo.dataFinal)
  <= 90
```

A expressão OCL da condição de guarda *[dr1]* é a negação da **Regra 1** e indica que se for verdadeira, o fluxo do processo deve seguir para o nó *dr1*, que é um nó que leva ou ao final do processo ou a uma nova escolha de período. Já a condição de guarda *[m2]* contém a expressão OCL que reflete a **Regra 1**, portanto indica que se for verdadeira, o fluxo do processo deve seguir para o nó *m2*, que está no caminho do fluxo principal, habilitando a execução da atividade *Selecionar Carro*.

Decisão d2

```
[dr2] process.carro.quilometragem > 50000
[Preparar Contrato] process.carro.quilometragem <= 50000
```

A expressão OCL da condição de guarda *[dr2]* é a negação da **Regra 6** e indica que se for verdadeira, o fluxo do processo deve seguir para o nó *dr2*, que é um nó que leva ou ao final do processo ou a uma escolha de um outro carro. Já a condição de guarda *[Preparar Contrato]* contém a expressão OCL que reflete a **Regra 6**, portanto indica que se for verdadeira, o fluxo do processo deve seguir para a atividade *Preparar Contrato*, que está no caminho do fluxo principal.

Decisão d3

```
[dr3] process.cliente.cartaoCredito.saldo <
  process.aluguel.precoTotal
  or (not process.cliente.cartaoCredito.validade.
  isAfter(process.periodo.dataFinal))
[dr4] process.cliente.cartaoCredito.saldo >=
  process.aluguel.precoTotal
  and process.cliente.cartaoCredito.validade.
  isAfter(process.periodo.dataFinal)
```

A expressão OCL da condição de guarda *[dr3]* é a negação da **Regra 2** e da **Regra 9** e indica que se for verdadeira, o fluxo do processo deve seguir para o nó *dr3*, que é um nó que leva ou ao final do processo ou a uma escolha de um outro cartão. Já a condição de guarda *[dr4]* contém a expressão OCL que reflete a **Regra 2** e a **Regra 9**, portanto indica que se for verdadeira, o fluxo do processo deve seguir para o nó *dr4*, que está no caminho do fluxo principal, habilitando a execução da próxima atividade.

5.2 Gerar Casos de Teste

Tendo em mãos todos os modelos e as regras necessários, devemos gerar os caminhos-chave, que servirão de base para a geração dos casos de teste e como critério de cobertura dos testes. Dado o diagrama da figura 5.2, a ferramenta PRUV gera seguintes caminhos-chave:

Caminho-Chave 1: Início ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ dr1 ⇒ Fim

Caminho-Chave 2: Início ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ Fim

Caminho-Chave 3: Início ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ Fim

Caminho-Chave 4: Início ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ m3 ⇒ Preparar Contrato ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ Fim

Caminho-Chave 5: Início ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ dr1 ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ Fim

Caminho-Chave 6: Início ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ m3 ⇒ Preparar Contrato ⇒ Verificar Credito ⇒ d3 ⇒ dr4 ⇒ m5 ⇒ Assinar Contrato ⇒ Fim

Caminho-Chave 7: Início ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ m3 ⇒ Preparar Contrato ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ m3 ⇒ Preparar Contrato ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ Fim

Caminho-Chave 8: Início ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ m3 ⇒ Preparar Contrato ⇒ Verificar Credito ⇒ d3 ⇒ dr4 ⇒ m4 ⇒ Notificar Motorista Adicional ⇒ dr5 ⇒ m5 ⇒ Assinar Contrato ⇒ Fim

Caminho-Chave 9: Início ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ m3 ⇒ Preparar Contrato ⇒ Verificar Credito ⇒ d3 ⇒ dr4 ⇒ m4 ⇒ Notificar Motorista Adicional ⇒ dr5 ⇒ m4 ⇒ Notificar Motorista Adicional ⇒ dr5 ⇒ m5 ⇒ Assinar Contrato ⇒ Fim

Executando-se todos esses 9 caminhos-chaves garantimos a cobertura de 100% do diagrama. Encerrada a etapa de geração dos caminhos-chave, podemos iniciar a etapa de geração dos casos de teste. Devemos validar as expressões OCL existentes em

cada nó percorrido em um caminho-chave. As expressões devem ser transformadas até que restem apenas variáveis independentes, conforme descrito na seção 2.5. Não apenas as pré e pós-condições e condições de guarda pertencentes ao caminho-chave em questão devem ser avaliados, mas também as invariantes das classes.

Para o caminho-chave 1, temos as seguintes regras: O nó atividade *Selecionar Periodo* tem as pós-condições periodo.dataInicial = dataInicio, periodo.dataFinal = dataFim, *periodo.dataFinal.isAfter(periodo.dataInicial)* e periodo.dataInicial.isAfter(now()) or periodo.dataInicial.isEqual(now()). O próximo nó é a decisão *d1*, com as condições de guarda [dr1] periodo.dataInicial.duration(periodo.dataFinal) > 90 que, se for verdadeira, leva ao nó *dr1*, e [m2] periodo.dataInicial.duration(periodo.dataFinal) <= 90, que leva ao nó *m2* caso seja verdadeira. Para o caminho-chave 1, temos que percorrer o nó *dr1*, portanto, escolhemos a primeira condição de guarda do nó *d1*.

Assim, pegando como base o caminho-chave 1, temos as seguintes condições: periodo.dataInicial.duration(periodo.dataFinal) > 90, *periodo.dataInicial = dataInicio*, periodo.dataFinal = dataFim, periodo.dataFinal.isAfter(periodo.dataInicial) e também periodo.dataInicial.isAfter(now()) or periodo.dataInicial.isEqual(now()), das quais podemos extrair as seguintes variáveis: *periodo.dataInicial* e *periodo.dataFinal*, que são atributos de classes e *dataInicio* e *dataFim*, que são parâmetros de entrada. De acordo com o teorema 2.5.1 e a tabela 2.5.1, podemos classificar essas variáveis como está a tabela 5.2.

Tabela 5.2: Classificação das variáveis presentes no caminho-chave 1

Variáveis Dependentes Ativas	**Variáveis Independentes Passivas**
periodo.dataInicial	datainicio
periodo.dataFinal	dataFim

Podemos usar as variáveis independentes para achar as partições de valores para as variáveis dependentes, realizando transformações nas condições. A tabela 5.3 apresenta as transformações para as condições presentes no caminho-chave 1. A primeira

coluna da tabela indica o número da condição, usado apenas como referência para mostrar quais condições são usadas nas transformações. Caso a condição seja uma transformação de outras, a terceira coluna indica quais foram as condições transformadas para originar esta. Para facilitar a visualização, iremos substituir as funções isAfter e isEqual pelos símbolos "$>$" e "$=$", respectivamente.

Tabela 5.3: Transformação das condições presentes no caminho-chave 1

Nº da Condição	**Condição**	**Condições usadas para transformações**
1	periodo.dataInicial.duration(periodo.dataFinal) $>$ 90	–
2	periodo.dataInicial = dataInicio	–
3	periodo.dataFinal = dataFim	–
4	periodo.dataFinal $>$ periodo.dataInicial	–
5	periodo.dataInicial = now()	–
6	dataInicio.duration(dataFim) $>$ 90	1, 2, 3
7	dataInicio = now()	4, 2, 3

A partir das condições de 1 a 5 foi possível chegar às condições 6 e 7 na tabela 5.3. Sabendo que now() é uma função que retorna a data atual, podemos considerar essa expressão como uma variável independente. A expressão dataInicio.duration(dataFim) retorna o intervalo de dias entre *dataInicio* e *dataFim*. Sendo assim, as condições 6 e 7 contém novas restrições para os intervalos de valores dos parâmetros de entrada *dataInicio* e *dataFim*. Ou seja, *dataInicio* deve ser igual à data atual e *dataFim* deve ser 90 dias maior que *dataInicio*.

Considerando que a data atual seja 16/06/2010, os intervalos de valores possíveis são *dataInicio* = 16/06/2010 e *dataFim* $>$ 13/09/2010. Devem ser gerados casos de teste com valores de fronteira, valores próximos à fronteira e valores aleatórios dentro do intervalo de valores. Com o uso da linguagem ASSL, a ferramenta PRUV é capaz de gerar inúmeros valores quem atendam a esse critério. Valores fora dos intervalos irão gerar, durante a execução do processos, caminhos diferente dos esperados nos

caminhos-chave e com isso não se pode garantir a cobertura de 100% do diagrama de atividades.

Com isso já conseguimos gerar valores para a execução dos nós *Selecionar Periodo* e *d1*. Mas *dr1* é uma decisão, e é necessário escolher uma transição de saída para este nó. Entretanto *dr1* não é uma decisão computável, ou seja, não há nenhum cálculo a ser feito para determinar qual será a transição de saída correta. Neste diagrama, *dr1* representa uma escolha aleatória. Dessa forma, *dr1* não possui condições de guarda. Como sabemos que de acordo com o caminho-chave 1 o próximo nó é o *Fim*, então basta prosseguir pela transição *dr1*⇒*Fim* e assim encerramos a geração de casos de teste para o caminho-chave 1. Os procedimentos para a geração dos casos de teste para os demais caminhos-chave são análogos a estes descritos aqui.

5.3 Executar os Casos de Teste

Esta etapa consiste na simulação do processo, ou seja, na execução dos casos de testes gerados na etapa anterior. Para realizar essa simulação executamos os algoritmos descritos na seção 3.3.4, percorrendo todos os nós de acordo com os parâmetros de entrada gerados no caso de teste.

Durante a simulação há duas situações possíveis:

1. a simulação consegue alcançar o nó final do diagrama de atividades, completando sua execução com sucesso (como foi o caso do caminho-chave 1), supondo que para aquele cenário o modelo de processo está de acordo com as regras ou a simulação ou
2. em um determinado momento, encontra uma violação do cenário com as regras de negócio.

5.3.1 Exemplo: Execução do Caminho-Chave 1

A simulação de um caso de teste do caminho-chave 1 acontece da seguinte maneira: O algoritmo irá passar pelos nós *Início* e *m1*, que não têm nenhuma ação, até chegar a *Selecionar Periodo*. Este nó é uma atividade, e a execução de suas ações está descrita no procedimento selecionarPeriodo(dataInicio:Date,dataFim:Date). Alguns valores para os 2 parâmetros de entrada desse procedimento já foram gerados. Vamos pegar o caso de teste onde $dataInicio = 16/06/2010$ e $dataFim = 14/09/2010$. A execução dessa atividade irá gerar os objetos do cenário mostrado na figura 5.3.

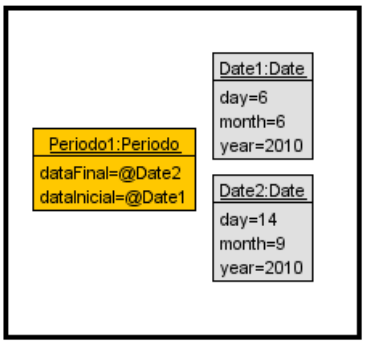


Figura 5.3: Objetos após a execução de Selecionar Período

Como já descrito na seção 5.2, após *Selecionar Periodo*, temos o nó *d1*, que de acordo com o caso de teste, irá selecionar *dr1* como próximo nó. O nó *dr1* irá selecionar o nó *Fim*, chegando, assim, ao fim da execução deste caso de teste sem falhas.

A execução de todos os outros cenários para todos os caminhos-chave é análoga a esta, pois os algoritmos utilizados servem para qualquer tipo de caminho. O que garante que a simulação do processo irá seguir o caminho-chave determinado são os casos de teste gerados para tal caminho. Dessa forma, se os casos de teste não forem gerados da maneira correta, não será garantido que o caminho-base especificado será percorrido.

A ferramenta PRUV não força a execução de um caminho. Ela vai seguindo o caminho de acordo com as condições de guarda, que são avaliadas em tempo real. Dessa forma, a simulação de um cenário do processo se aproxima mais de uma execução real.

5.3.2 Exemplo: Execução do Caminho-Chave 4

Vamos proceder com a simulação de um caso de teste para o caminho-chave 4. Dados os parâmetros de entrada, a simulação iniciará e prosseguirá até atingir o nó *m3*. Neste ponto, o cenário dos objetos do processo está de acordo com a figura 5.4(a). Ao tentar executar o próximo nó, que é a atividade *Preparar Contrato*, um erro é encontrado. Neste caso, a ferramenta *PRUV* é capaz de informar o caminho percorrido até encontrar o erro e qual foi a regra violada, exibindo a seguinte mensagem:

Total time up to this node: 6.181422373371013
Path to reach this node: Start –> m1 -> Selecionar Periodo –> d1 -> m2 –> Selecionar Carro –> d2 –> m3 –> Preparar Contrato
Error found is: PRE-CONDITION FALSE: precondition 'PrepararContrato_pre2' is false

Neste momento o usuário deve analisar porque o erro ocorreu. A pré-condição *PrepararContrato_pre2* espera que o atributo *process.carro.situacao* seja igual a *'disponivel'*, porém a pós-condição da atividade *Selecionar Carro*, que é a atividade imediatamente anterior a esta, espera que o valor deste atributo seja *'selecionado'*. Então existe aí um conflito de regras. Neste caso o analista poderia perceber que a pré-condição *PrepararContrato_pre2* não deveria existir, já que após executar a atividade *Selecionar Carro* o carro utilizado como entrada na atividade *Preparar Contrato* fica marcado como *'selecionado'*.

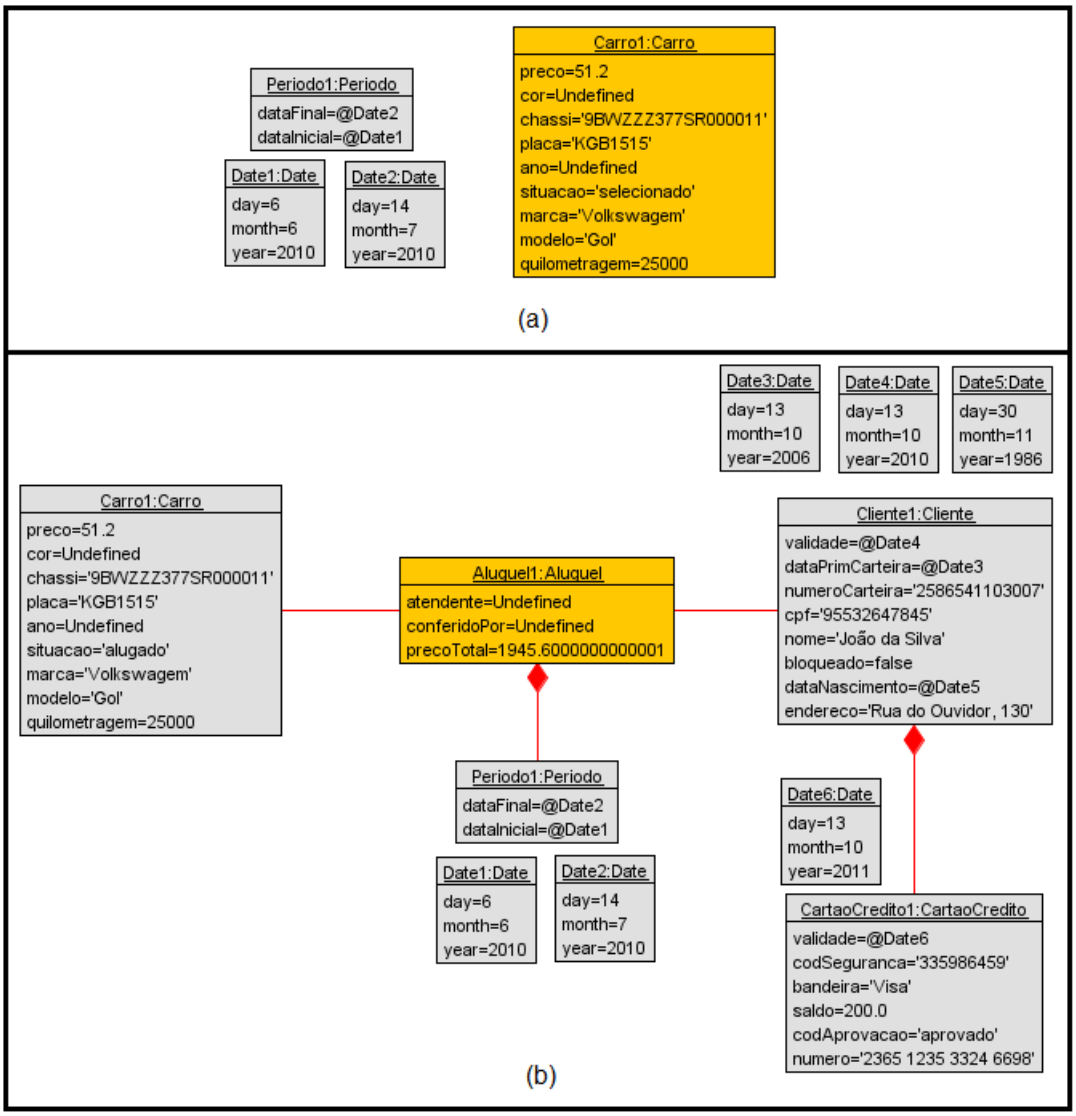


Figura 5.4: Cenários antes e após a correção dos modelos no caminho-chave 4

Após o erro ser detectado e corrigido, os casos de teste devem ser gerados e executado novamente para cada caminho-chave, a fim de verificar se serão executados com sucesso. Após as correções sugeridas acima para o erro encontrado, o novo caso de teste para o caminho-chave 4 foi executado, desta vez com sucesso, criando o cenário de objetos da figura 5.4(b)

5.3.3 Exemplo: Execução do Caminho-Chave 8

Para o caminho-chave 8, a simulação ocorre normalmente até a execução do nó *m5*. Neste ponto da simulação, o cenário dos objetos está de acordo com a figura 5.5. O próximo nó do caminho é *Assinar Contrato*. Ao tentar executar esse nó um erro é encontrado e a ferramenta PRUV nos fornece a seguinte mensagem:
Total time up to this node: 23.526729422523417
Path to reach this node: Start —> m1 —> Selecionar Periodo —> d1 —> m2 —> Selecionar Carro —> d2 —> m3 —> Preparar Contrato —> Verificar Credito —> d3 —> dr4 —> m4 —> Notificar Motorista Adicional —> dr5 —> m5 —> Assinar Contrato
Error found is: PRE-CONDITION FALSE: precondition 'assinarContrato_pre01' is false

O erro ocorreu na pré-condição *assinarContrato_pre01*, cuja expressão é process.cartaoCredito.saldo >= process.aluguel.precoTotal. O erro ocorreu porque a atividade *Notificar Motorista Adicional* aumenta o valor do atributo *process.aluguel.precoTotal*, mas o saldo do cartão de crédito só é conferido na atividade *Verificar Credito*, que é executada antes deste atributo ser modificado. Portanto, uma possível correção ou melhoria no processo é alterar a ordem em que *Verificar Credito* é executada, passando-a para depois de *Notificar Motorista Adicional*, já que esta última atividade modifica o valor total do aluguel. O resultado dessa alteração é mostrado na figura 5.6.

Como a estrutura do modelo de processo foi alterada, os caminhos-chave devem ser alterados, e novos casos de teste devem ser gerados para estes caminhos. Os novos caminhos-chave gerados na PRUV foram:

Caminho-Chave 1:
Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ dr1 ⇒ Fim

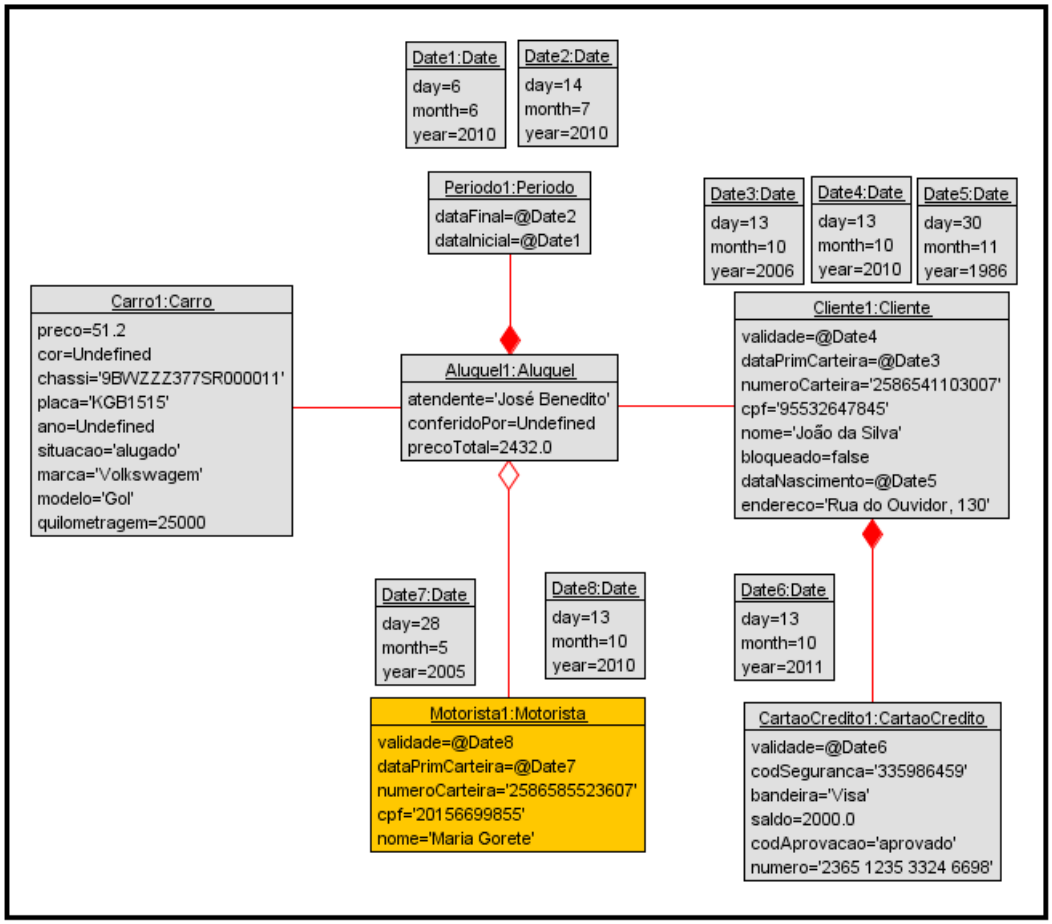


Figura 5.5: Cenários do caminho-chave 8 após a execução de Notificar Motorista Adicional

Caminho-Chave 2:

Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ Fim

Caminho-Chave 3:

Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ Fim

Caminho-Chave 4:

Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ dr1 ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ Fim

Caminho-Chave 5:
Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ Preparar Contrato ⇒ dr4 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ Fim

Caminho-Chave 6:
Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ Preparar Contrato ⇒ dr4 ⇒ m4 ⇒ Notificar Motorista Adicional ⇒ dr5 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ Fim

Caminho-Chave 7:
Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ Preparar Contrato ⇒ dr4 ⇒ m4 ⇒ Notificar Motorista Adicional ⇒ dr5 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ Assinar Contrato ⇒ Fim

Caminho-Chave 8:
Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ Preparar Contrato ⇒ dr4 ⇒ m4 ⇒ Notificar Motorista Adicional ⇒ dr5 ⇒ m4 ⇒ Notificar Motorista Adicional ⇒ dr5 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ Fim

Caminho-Chave 9:
Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ Preparar Contrato ⇒ dr4 ⇒ m4 ⇒ Notificar Motorista Adicional ⇒ dr5 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ Fim

5.3.4 Exemplo: Execução do Caminho-Chave 8 após correção

Como após a alteração anterior novos caminhos-chave e casos de teste foram gerados, temos que realizar a execução destes novos caminhos. Como exemplo, iremos executar um caso de teste para o novo caminho-chave 8 gerado. Observamos que este caminho chave contém um *loop*, que engloba o trecho *m4* $\Rightarrow$ *Notificar Motorista Adicional* $\Rightarrow$ *m3* $\Rightarrow$ *dr5* $\Longrightarrow$ *m4* $\Rightarrow$ *Notificar Motorista Adicional* $\Rightarrow$ *m3* $\Rightarrow$ *dr5*. Portanto, para testes mais apurados, podemos repetir um determinado número de vezes esse trecho em um caso de testes. Neste exemplo, vamos executá-lo 3 vezes. Vamos partir do ponto onde a atividade *Notificar Motorista Adicional* já tenha sido executada 2 vezes e que o cenário de objetos esteja de acordo com a figura 5.7.

Ao executar a atividade *Notificar Motorista Adicional* pela terceira vez, uma violação é encontrada, causando o erro exibido pela ferramenta PRUV:
Path to reach this node: Start —> m1 —> Selecionar Periodo —> d1 —> m2 —> Selecionar Carro —> d2 —> Preparar Contrato —> dr4 —> m4 —> Notificar Motorista Adicional —> dr5 —> m4 —> Notificar Motorista Adicional —> dr5 —> m4 —> Notificar Motorista Adicional
Error found is: PRE-CONDITION FALSE: precondition 'Max2Adicionais' is false
evaluation results:
self : Process = @process
self.aluguel : Aluguel = @Aluguel1
self.aluguel.motoristaAdicional : Set(Motorista) = Set{@Motorista1, @Motorista2, @Motorista3}
self.aluguel.motoristaAdicional->size : Integer = 2
2 : Integer = 2
(process.aluguel.motoristaAdicional->size() < 2) : Boolean = false

Esta mensagem nos indica que a pré-condição *Max2Adicionais*, que afirma que a quantidade de motoristas adicionais associados ao aluguel deve ser menor ou 2 antes da execução da atividade foi violada, pois um segundo motorista já existia no aluguel.

Este erro ocorreu pois a atividade *Notificar Motorista Adicional* está em um *loop*, podendo ser executada de zero até um número infinito de vezes. Além de ser possível adicionar apenas 2 motoristas, podemos observar que o valor do aluguel sofre um acréscimo apenas para adicionar o primeiro motorista. Se a regra que diz que apenas são permitidos dois motoristas adicionais não existisse, poderiam ser adicionados quantos motoristas quanto desejados, e o valor do aluguel poderia ser recalculado para cada motorista adicional. Caso não seja permitido mudar as regras de negócio, a alteração pode ser realizada no processo. O modelo de processo pode ser alterado de tal forma que não seja possível adicionar mais de dois motoristas. Esta correção pode ser vista na figura 5.8

Para realizar uma melhoria no processo, a atividade *Notificar Motorista Adicional* foi substituída por duas: *Notificar Motorista Adicional 1* e *Notificar Motorista Adicional 2*. As regras associadas a estas duas novas atividades são:

Notificar Motorista Adicional 1

```
pre NotificarMotoristaAdicional1_pre1:
  process.aluguel.motoristaAdicional->size() = 0
procedure notificarMotoristaAdicional1(umMotorista: Motorista)
begin
  Insert(Motorista_Aluguel, [process.aluguel], [umMotorista]);
  [process.aluguel].precoTotal := [process.aluguel.precoTotal* 1.25];
  return umMotorista;
end;
```

```
post NotificarMotoristaAdicional1_post01:
  process.aluguel.precoTotal = self.aluguel.precoTotal@pre* 1.25
post NotificarMotoristaAdicional1_post02:
  process.aluguel.motoristaAdicional->size() = 1
```

Notificar Motorista Adicional 2

```
pre NotificarMotoristaAdicional2_pre01:
  process.aluguel.motoristaAdicional->size() = 1

procedure notificarMotoristaAdicional2(umMotorista: Motorista)
begin
  Insert(Motorista_Aluguel, [process.aluguel], [umMotorista]);
  return umMotorista;
end;

post NotificarMotoristaAdicional2_post01:
  process.aluguel.motoristaAdicional->size() = 2
```

Para adequar os testes a estas modificações, os caminhos-chave devem ser gerados novamente, e para estes novos caminho-chave devem ser gerados novos casos de teste. A seguir mostramos quais foram os novos caminhos-chave gerados pela ferramenta PRUV:

Caminho-Chave 1:
Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ dr1 ⇒ Fim

Caminho-Chave 2:
Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ Fim

Caminho-Chave 3:

Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ Fim

Caminho-Chave 4:

Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ dr1 ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ dr2 ⇒ Fim

Caminho-Chave 5:

Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ Preparar Contrato ⇒ dr4 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ Fim

Caminho-Chave 6:

Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ Preparar Contrato ⇒ dr4 ⇒ Notificar Motorista Adicional 1 ⇒ dr5 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ Fim

Caminho-Chave 7:

Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ Preparar Contrato ⇒ dr4 ⇒ Notificar Motorista Adicional 1 ⇒ dr5 ⇒ Notificar Motorista Adicional 2 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ Fim

Caminho-Chave 8:

Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒ Preparar Contrato ⇒ dr4 ⇒ Notificar Motorista Adicional 1 ⇒ dr5 ⇒ Notificar Motorista Adicional 2 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ Assinar Contrato ⇒ Fim

Caminho-Chave 9:

Inicio ⇒ m1 ⇒ Selecionar Periodo ⇒ d1 ⇒ m2 ⇒ Selecionar Carro ⇒ d2 ⇒

Preparar Contrato ⇒ dr4 ⇒ Notificar Motorista Adicional 1 ⇒ dr5 ⇒ Notificar Motorista Adicional 2 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ m3 ⇒ Verificar Credito ⇒ d3 ⇒ dr3 ⇒ Fim

Os casos de teste para os novos caminhos-chave gerados a partir das correções tanto feitas nas regras quanto na estrutura dos modelos, dessa vez conseguem rodar com sucesso, indicando que os erros encontrados anteriormente não existem mais.

Este exemplo mostrou que o método proposto pode ser usado não só para apontar inconsistências entre as regras e os processos de negócio ou entre as próprias regras de negócio. O método também é capaz de ajudar os analistas a identificarem pontos onde o processo ou as regras podem ser melhorados, fazendo, assim, com que a execução do processo tenha melhores resultados.

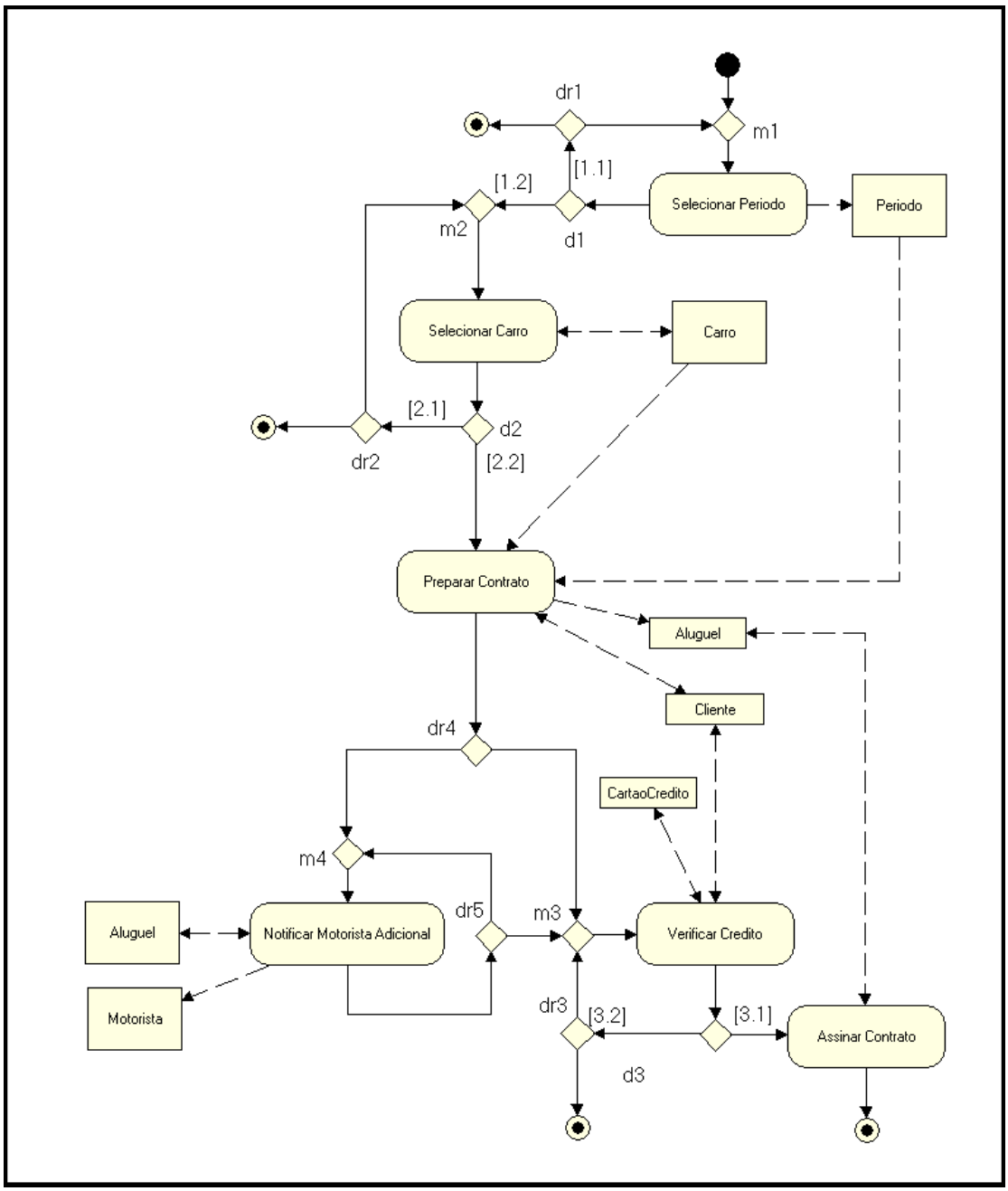


Figura 5.6: Correção na ordem da atividade Verificar Credito

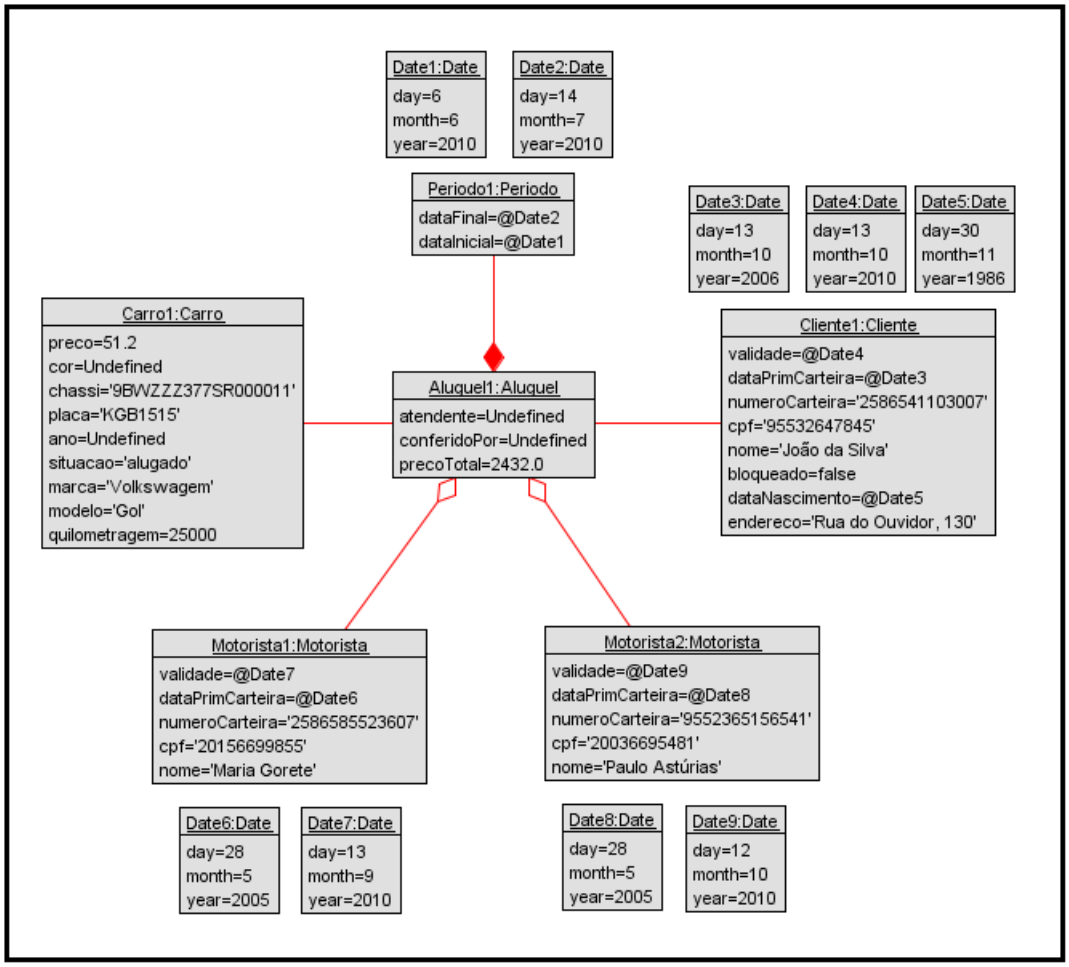


Figura 5.7: Cenário da falha encontrada no caminho-chave 8

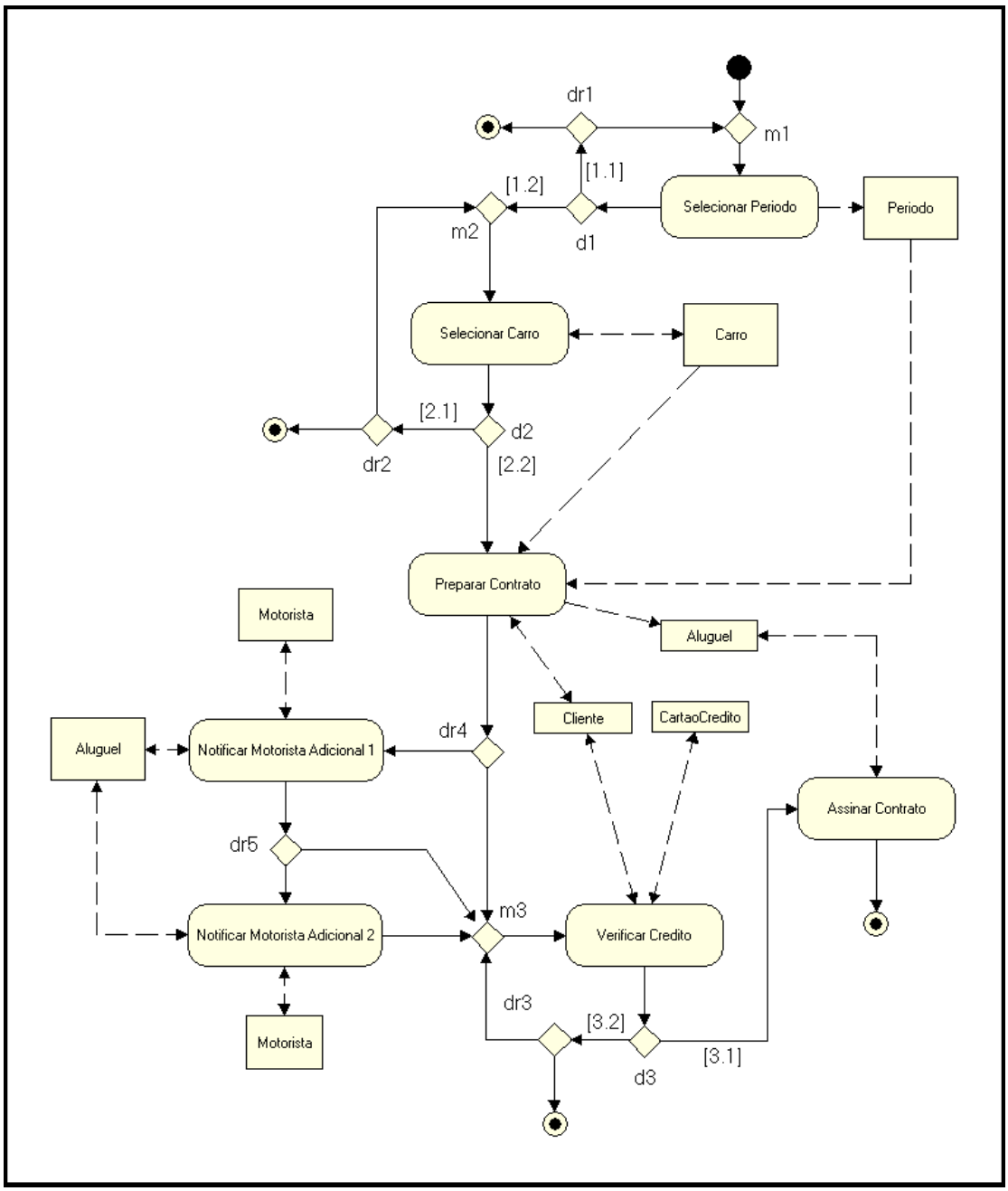


Figura 5.8: Segunda correção no modelo de processo

6 CONSIDERAÇÕES FINAIS

6.1 Resumo dos resultados

A verificação de conformidade de processos de negócio com regras de negócio é uma fonte importante de vantagem competitiva atualmente. O método proposto nesta dissertação provê tanto ao analista de negócio quanto aos usuários do processo uma ferramenta para a validação da adequação de novos processos de negócios com as políticas do negócio.

Este método propicia também uma forma sistemática para a simulação baseada nos cenários de testes. Apesar da simulação repetitiva do processo não garantir a conformidade total, ela dá uma garantia de que o processo não falhará nas situações mais elementares. O mesmo não pode ser dito de métodos *ad hoc*.

Muitas abordagens para a geração de cenários de testes baseiam-se em apenas gerar dados de testes aleatórios e medir a cobertura desejada. No entanto, o método desenvolvido por Weißleder e Schlingloff (2007a; 2007b), que foi adotado e adaptado para o método apresentado neste trabalho não é meramente aleatório. Além disso, o método mede a porcentagem do critério de cobertura ao invés de forçá-la. A ferra-

menta PRUV gera todos os caminhos-chave necessários para que 100% do diagrama de atividades (incluindo seus nós e transições) seja coberto, mas não força esses caminhos. Os caminhos vão se formando de acordo com os casos de teste dados, e as decisões, condições de guarda, pré e pós condições e invariantes são calculados dinamicamente, o que torna a simulação mais próxima de uma execução real. Além disso, a cada passo da execução, as transições e nós percorridos são marcados, assim a ferramenta PRUV é capaz de medir qual a porcentagem da cobertura em tempo real.

Através do exemplo apresentado no capítulo 5, mostramos que este método é capaz de detectar diferentes tipos de erros, que podem ocorrer tanto nas regras de negócio quanto no modelo de processo. Quando há uma violação à conformidade, o cenário, o caminho percorrido no processo e as regras envolvidas são evidenciados, o que torna mais fácil a depuração e correção da falha de conformidade. Isso implica em um modelo mais consistente e confiável, já que a atividade de validação se inicia no estágio de modelagem.

6.2 Contribuições

Diferentes abordagens para a verificação automática de conformidade de processos têm se apresentado na literatura. Por exemplo, a técnica de diagrama de estados (MUTH et al., 1998), modelos baseados em rede de Petri (AALST, 2003), o uso de π-calculus (YANG; ZHANG, 2003), lógica temporal (ESHUIS; WIERINGA, 2001), entre outras. Entretanto, todas essas abordagens requerem, em um certo ponto, a prova formal de uma ou mais proposições, que não são facilmente alcançadas, fazendo seu uso uma tarefa muito difícil para um gerente comum, que geralmente tem um conhecimento restrito de lógica matemática e métodos de prova.

Este trabalho oferece como principal contribuição um método capaz de validar a conformidade de processos de negócio com regras de negócio de modo mais simples que técnicas formais existentes e que pode ser utilizado sem que seja necessário um profundo conhecimento de lógica matemática para provas formais.

O método é ainda capaz de realizar uma combinação de diferentes critérios de cobertura, como o critério baseado em valores de fronteira e o critério baseado em fluxo de controle, preenchendo assim lacunas deixadas quando se usa apenas um método isolado (WEIßLEDER; SCHLINGLOFF, 2008).

Como contribuição também foi desenvolvida a ferramenta PRUV, que automatiza este método. A ferramenta usa como entrada os diagramas UML, as regras OCL e os casos de teste para realizar a execução do processo. Durante a execução dos casos de teste a ferramenta valida, a cada passo do andamento do processo, as regras de negócio. Desta forma, a ferramenta é capaz de parar o processo no momento exato em que uma regra é violada, mostrando quais as etapas realizadas dentro do processo, o cenário dos objetos no sistema e precisamente qual regra foi violada, indicando qual era o valor esperado e qual foi o valor encontrado. Além do mais é possível exportar todos os passos executados pela PRUV para a ferramenta USE. Dessa forma, é possível visualizar dentro da USE um diagrama de objetos e um diagrama de sequência, para cada passo executado na PRUV. Com isso, o entendimento e correção das falhas tornam-se mais fáceis.

Além disso, a introdução de novos processos de negócio e a atualização dos existentes têm um pequeno impacto no processo de validação, tornando o método numa opção robusta e confiável tanto para gerentes quanto para analistas de processos de negócio.

Outro resultado alcançado foi a publicação do artigo "*A method for validating the compliance of business processes to business rules*" (ARAUJO et al., 2010), apresen-

tado no congresso *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, que aconteceu nos dias 22 a 26 de março de 2010 em Sierre, Suíça.

6.3 Limitações

A validação através de simulações não garante a consistência entre os processos e as regras de negócio, mas permite obter um determinado nível de confiança. O método para a geração dos cenários é capaz de cobrir 100% do diagrama, mas ainda é necessário tratar a questão de como atingir um nível de confiança desejado.

Além disso, as ferramentas na qual a PRUV foi baseada não contemplam toda a estrutura definida pela UML, mas somente aquelas que são utilizadas com maior frequência e que se mostram suficientes durante o desenvolvimento do método.

Outra limitação está na falta de um experimento com o método sendo utilizado em situações reais e por um número significativo de pessoas para se obter dados estatísticos quanto à facilidade e agilidade e benefícios no uso deste método.

6.4 Trabalhos Futuros

Nossos esforços estão concentrados em algumas questões importantes, tais como: quantos cenários precisam ser simulados para garantir um nível mínimo de confiança, como gerar automaticamente as instâncias dos cenários de teste e como expandir este método para modelos de processos de negócio mais genéricos. Atualmente a ferramenta PRUV gera automaticamente apenas os caminhos-chave usados nos casos de teste.

Além disso, pretendemos implantar o método em um ambiente empresarial real nos processos onde a empresa percebe que os resultados do processo não são os esperados. Através da simulação do processo poderemos descobrir se alguma das regras de negócio é violada durante execução do processo e assim identificar as causas do problema. Ao corrigir os modelos e acabar com as falhas encontradas na simulação, as regras e o processo a empresa poderá ajustar seu funcionamento de acordo com as alterações realizadas, corrigindo, assim, as falhas encontradas. Também espera-se que além de encontrar falhas, o método possa apontar possíveis melhorias que nos processos de negócio.

REFERÊNCIAS

AALST, W. M. P. van der. Challenges in business process management: verification of business processes using petri nets. **Bulletin of the EATCS**, [S.l.], v.80, p.174–199, 2003. European Association for Theoretical Computer Science (EATCS).

ANNAS, G. J. HIPAA Regulations - A New Era of Medical-Record Privacy? **The New England Journal of Medicine**, [S.l.], v.348, n.15, p.1486–1490, Abril 2003.

ARAUJO, B. M.; SCHMITZ, E. A.; CORREA, A. L.; ALENCAR, A. J. A method for validating the compliance of business processes to business rules. In: SAC '10: PROCEEDINGS OF THE 2010 ACM SYMPOSIUM ON APPLIED COMPUTING, 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.145–149.

ARMSTRONG, T. K. Digital Rights Management and the Process of Fair Use. **Harvard Journal of Law & Technology**, [S.l.], v.20, n.1, p.49–121, Outono 2006.

BARBALHO, S. C. M.; ROZENFELD, H.; AMARAL, D. C. Modelando Processos de Negócio com UML. In: XXII ENCONTRO NACIONAL DE ENGENHARIA DE PRODUÇÃO, 2002, Curitiba, PR, Brasil. **Anais...** [S.l.: s.n.], 2002.

BARJIS, J. A Business Process Modeling and Simulation Method Using DEMO. In: ENTERPRISE INFORMATION SYSTEMS, 9TH INTERNATIONAL CONFERENCE - ICEIS (SELECTED PAPERS), 2007, Funchal, Madeira. **Proceedings...** Springer, 2007. p.254–265. (Lecture Notes in Business Information Processing).

BARJIS, J.; SHISHKOV, B.; DIETZ, J. L. Validation of business components via simulation. In: INTERNATIONAL EUROSIM 2001 CONGRESS, 4., 2001. **Proceedings...** [S.l.: s.n.], 2001.

BLANCO, R.; TUYA, J.; ADENSO-DÍAZ, B. Automated test data generation using a scatter search approach. **Information and Software Technology**, Lillehammer - Norway, v.51, n.4, 2009.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)**. [S.l.]: Addison-Wesley Professional, 2005.

CHEN, M.; MISHRA, P.; KALITA, D. Coverage-driven Automatic Test Generation for UML Activity Diagrams. **GLSVLSI**, [S.l.], p.139–142, 2008.

CIMATTI, A.; CLARKE, E.; GIUNCHIGLIA, E.; GIUNCHIGLIA, F.; PISTORE, M.; ROVERI, M.; SEBASTIANI, R.; TACCHELLA, A. NuSMV Version 2: an opensource tool for symbolic model checking. In: PROC. INTERNATIONAL CONFERENCE ON COMPUTER-AIDED VERIFICATION (CAV 2002), 2002, Copenhagen, Denmark. **Proceedings...** Springer, 2002. (LNCS, v.2404).

CORREA, A. L. **Reestruturando Especificações de Restrições de Modelos Elaboradas em OCL**. 2006. Tese (Doutorado em Ciência da Computação) — COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil.

CUNHA, C. A. A. **Uma abordagem para a transformação de regras de negócio na arquitetura dirigida por modelos**. 2009. Dissertação (Mestrado em Ciência da Computação) — Programa de Pós-graduação em Informática - Núcleo de Computação Eletrônica - Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil.

DECAMPS, J.-P.; ROCHET, J.-C.; ROGER, B. The three pillars of Basel II: opti-

mizing the mix. **Journal of Financial Intermediation**, [S.l.], v.13, n.2, p.132–155, Abril 2004.

DESEL, J.; JUHÁS, G. "What Is a Petri Net?" Informal Answers for the Informed Reader. **Lecture Notes in Computer Science**, Berlin / Heidelberg, v.2128, n.4, p.1–25, 2001.

ESHUIS, R.; WIERINGA, R. A Real-Time Execution Semantics for UML Activity Diagrams. In: PROCEEDINGS OF THE 4^{th} INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING (FASE), 2001, Genova, Italy. **Proceedings...** Springer-Verlag, 2001. p.76–90. Lecture Notes in Computer Science 2029.

GOGOLLA, M.; BÜTTNER, F.; RICHTERS, M. Validating UML and OCL models in USE by Automatic Snapshot Generation. **Journal on Software and System Modeling**, [S.l.], v.4, p.2005, 2005.

GOGOLLA, M.; BÜTTNER, F.; RICHTERS, M. USE: a uml-based specification environment for validating uml and ocl. **Science of Computer Programming**, [S.l.], v.69, p.27–34, Dezembro 2007.

HALVEY, J. K.; MELBY, B. M. **Business Process Outsourcing**: process, strategies, and contracts. 2^{a}.ed. [S.l.]: Wiley, 2007.

JANSEN-VULLERS, M.; NETJES, M. Business Process Simulation - Tool Survey. In: THE SEVENTH WORKSHOP ON THE PRACTICAL USE OF COLOURED PETRI NETS AND CPN TOOLS, 2006, Aarhus. **Proceedings...** K. Jensen. - University of Aarhus, 2006. p.77–96.

KELLNER, M. I.; MADACHY, R. J.; RAFFO, D. M. Software process simulation modeling: why? what? how? **Journal of Systems and Software**, [S.l.], v.46, n.2-3, p.91 – 105, 1999.

KHARBILI, M. E.; MEDEIROS, A. de; STEIN, S.; AALST, W. M. van der. Business process compliance checking: current state and future challenges. In: MODELLING BUSINESS INFORMATION SYSTEMS, 2008, Saarbrücken, Germany. **Proceedings...** [S.l.: s.n.], 2008. p.107–113.

SCHEER, A. W.; ABOLHASSAN, F.; JOST, W.; KIRCHMER, M. (Ed.). **Business Process Automation**. 1ª.ed. [S.l.]: Springer, 2004. p.1–16.

KLEPPE, A. G.; WARMER, J.; BAST, W. **MDA Explained**: the model driven architecture: practice and promise. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

LALIOTI, V. Animation for Validation of Business System Specifications. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 1997, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 1997. v.2.

LIU, Y.; MÜLLER, S.; XU, K. A static compliance-checking framework for business process models. **IBM Systems Journal**, [S.l.], v.46, n.2, p.335 – 361, 2007.

LONGLEY, D.; BRANAGAN, M.; CAELLI, W. J.; KWOK, L. F. Feasibility of Automated Information Security Compliance Auditing. In: PROCCEDINGS OF THE 23^{rd} INTERNATIONAL INFORMATION SECURITY CONFERENCE, 2007, Cracow, Poland. **Proceedings...** Springer, 2007. v.278, p.493 – 507.

MADDEN, B. J. For Better Corporate Governance, The Shareholder Value Review. **Journal of Applied Corporate Finance**, [S.l.], v.19, n.1, p.102–115, Inverno 2007.

MEYER, B. **Object-Oriented Software Construction**. 2ª.ed. [S.l.]: Prentice Hall, 1997. 1254p.

MORGADO, G. P. **RAPDIS**: um processo e um ambiente MDA para o desenvolvimento de sistemas de informação. 2007. Dissertação (Mestrado em Ciência da

Computação) — Programa de Pós-graduação em Informática - Núcleo de Computação Eletrônica - Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil.

MORGAN, T. **Business Rules and Information Systems**: aligning with business goals. [S.l.]: Addison-Wesley, 2002. ISBN: 0-201-74391-4.

MUEHLEN, M. zur; INDULSKA, M.; KAMP, G. Business process and business rules modeling languages for compliance management: a representational analysis. In: PROCEEDINGS OF THE 26^{th} INTERNATIONAL CONFERENCE ON CONCEPTUAL MODELING, 2007, Auckland, New Zealand. **Proceedings...** ACM Press, 2007. v.83, p.127 – 132.

LEONID, K.; DOGAC, A.; OZSU, M. T.; SHETH, A. P. (Ed.). **Workflow Management Systems and Interoperability**. [S.l.]: Springer, 1998. p.281–303. (NATO ASI Series).

NAMIN, A. S.; ANDREWS, J. H. The influence of size and coverage on test suite effectiveness. In: ISSTA '09: PROCEEDINGS OF THE EIGHTEENTH INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.57–68.

OMG Object Management Group. **OCL 2.0 Specification**. Disponível em: http://www.omg.org/cgi-bin/doc? ptc/2003-10-14.

OMG Object Management Group. **Unified Modeling Language (UML) Superstructure Specification, version 2.0**. Disponível em: http://www.omg.org/cgi-bin/doc? formal/05-07-04.

OMG Object Management Group. **SBVR 1.0 Specification**. Disponível em: http://www.omg.org/spec/SBVR/1.0/PDF/.

ORACLE. **The Java^{TM} Tutorials**. Disponível em: http://download.oracle.com/javase/tutorial/index.html.

PEREIRA, M. J. T. V. **Sistematização da Animação de Programas**. 2009. Tese (Doutorado em Ciência da Computação) — Departamento de Informática, Universidade do Minho, Minho, Portugal.

POOLE, J. **A Method to Determine a Basis Set of Paths to Perform Program Testing**. [S.l.]: National Institute of Standards and Technology, 1995. (5737).

PRESSMAN, R. S. **Engenharia de Software**. 6^a.ed. [S.l.]: McGraw-Hill Higher Education, 2006.

RIBEIRO, Ó. R. d. S. F. **Animation-based validation of reactive software systems using behavioural models**. 2009. Tese (Doutorado em Ciência da Computação) — Escola de Engenharia, Universidade do Minho, Minho, Portugal.

RICHTERS, M.; GOGOLLA, M. **ASSL - A Snapshot Sequence Language**. Disponível em http://www.db.informatik.uni-bremen.de/projects/USE/assl.ps.

ROMANO, R. The Sarbanes-Oxley Act and the making of quack corporate governance. **Yale Law Journal**, [S.l.], v.114, p.1521–1610, 2005.

ROSCA, D.; GREENSPAN, S.; WILD, C. Enterprise Modeling and Decision-Support for Automating the Business Rules Lifecycle. **Automated Software Engg.**, Hingham, MA, USA, v.9, n.4, p.361–404, 2002.

SADIQ, S.; GOVERNATORI, G.; NAMIRI, K. Modeling Control Objectives for Business Process Compliance. In: PROCEEDINGS OF THE 5^{th} INTERNATIONAL CONFERENCE ON BUSINESS PROCESS MANAGEMENT, 2007, Brisbane, Australia. **Proceedings...** Springer, 2007. p.149–164. Lecture Notes in Computer Science 4714.

SALAS, P. A. P.; AICHERNIG, B. K. **Automatic TestCase Generation for OCL: a mutation approach**. [S.l.: s.n.], 2005.

SCHNEIDER, J. Cost-of-Compliance Sweet Spot. **Strategic Finance**, [S.l.], p.27–31, Agosto 2007.

SCHROEDER, M. Verification of Business Processes for a Correspondence Handling Center Using CCS. In: PROCEEDINGS OF 5^{th} EUROPEAN SYMPOSIUM ON VALIDATION AND VERIFICATION OF KNOWLEDGE BASED SYSTEMS - THEORY, TOOLS AND PRACTICE, 1999, Deventer, The Netherlands. **Proceedings...** Wolters Kluwer, 1999. p.253–264.

SILVEIRA, D. S. da. **ANIMARE**: um método de validação dos processos de negócio através da animação. 2009. Tese (Doutorado em Ciência da Computação) — COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil.

SOKOLOWSKI, J. A.; BANKS, C. M. **Principles of Modeling and Simulation**: a multidisciplinary approach. [S.l.]: Wiley, 2009.

TSAI, A.; WANG, J.; TEPFENHART, W.; ROSCA, D. EPC workflow model to WIFA model conversion. In: IEEE INTERNATIONAL CONFERENCE ON SYSTEMS, MAN AND CYBERNETICS, 2006, Taipei. **Proceedings...** [S.l.: s.n.], 2006. p.2758–2763.

VARELLA, A.; PEREIRA, V.; VONHELD, V.; ZIMBRÃO, G.; SILVA, J. C. P. da. Uma Interface em Linguagem Natural para a Verificação de Regras de Negócio em Bases de Dados. In: IV SIMPÓSIO DE DESENVOLVIMENTO E MANUTENÇÃO DE SOFTWARE DA MARINHA, 2004, Rio de Janeiro, RJ, Brasil. **Anais...** [S.l.: s.n.], 2004.

WEIßLEDER, S.; SCHLINGLOFF, B.-H. Automatic Test Generation from Coupled UML Models using Input Partitions. In: PROCEEDINGS OF 4^{th} MODEVVA WORKSHOP ON MODEL-DRIVEN ENGINEERING, VERIFICATION AND VALIDATION, 2007, Nashville, TN, USA. **Proceedings...** [S.l.: s.n.], 2007. p.23–32. Best student paper award.

WEIßLEDER, S.; SCHLINGLOFF, B.-H. Deriving Input Partitions from UML Models for Automatic Test Generation. In: MODELS WORKSHOPS, 2007. **Proceedings...** Springer Berlin / Heidelberg, 2007. p.151–163. (Lecture Notes in Computer Science, v.5002).

WEIßLEDER, S.; SCHLINGLOFF, B.-H. Quality of Automatically Generated Test Cases based on OCL Expressions. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION, AND VALIDATION - ICST, 2008, Washington, DC, USA. **Proceedings...** EEE Computer Society, 2008. p.517–520.

WEIßLEDER, S.; SOKENOU, D. Automatic Test Case Generation from UML Models and OCL Expressions. In: SOFTWARE ENGINEERING (WORKSHOPS), 2008. **Proceedings...** GI, 2008. p.423–426. (LNI, v.122).

YANG, D.; ZHANG, S. Approach for workflow modeling using π-calculus. **Journal of Zhejiang University: SCIENCE**, [S.l.], v.4, n.6, p.643 – 650, 2003.

ZIMBRÃO, G.; MIRANDA, R. A.; SOUZA, J. M. de; NETO, F. P. FalaOCL: uma ferramenta para parafrasear OCL. In: XVI SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 2002, Gramado, RS, Brasil. **Anais...** [S.l.: s.n.], 2002.

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

CCMN - Bloco C - Cidade Universitária - Ilha do Fundão
Rio de Janeiro - RJ CEP: 21941-916
www.ppgi.ufrj.br