DIEGO LAVERSON FREITAS

FORMALIZAÇÃO DO DIAGRAMA DE CASOS DE USO POR MEIO DA LINGUAGEM EVENT-B

Monografia submetida ao Curso de Bacharelado em Ciência da Computação da Universidade Estadual do Piauí, como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação

Orientador: Prof. M.Sc. Thiago Carvalho de Sousa

PARNAÍBA

2013

DIEGO LAVERSON FREITAS

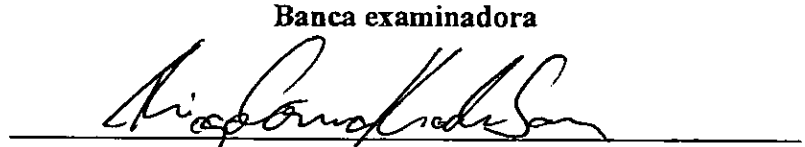
FORMALIZAÇÃO DO DIAGRAMA DE CASOS DE USO POR MEIO DA LINGUAGEM EVENT-B

Monografia submetida ao Curso de Bacharelado em Ciência da Computação da Universidade Estadual do Piauí, como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação

Orientador: Prof. M.Sc. Thiago Carvalho de Sousa

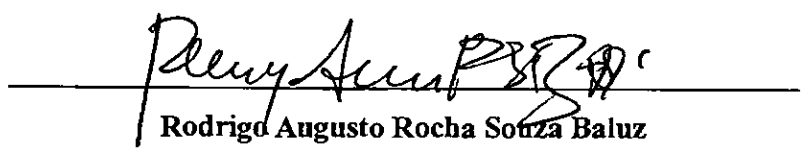
Aprovação em __/__/____

Banca examinadora



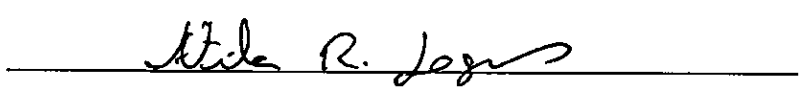
Thiago Carvalho de Sousa

Mestre



Rodrigo Augusto Rocha Souza Baluz

Mestre



Átila Rabelo Lopes

Mestre

DEDICATÓRIA

Dedico este trabalho a minha mãe, que foi minha maior incentivadora. Ela que sempre me deu forças para seguir em frente e conseguir alcançar meus objetivos.

A Deus, acima de tudo e de todos, por me conceder mais esta graça.

AGRADECIMENTO

Agradeço em primeiro lugar a Deus, que me permitiu chegar até aqui, e que sempre esteve ao meu lado.

A minha mãe Rosiane Rodrigues, que sempre me apoiou e me incentivou, não só nessa trajetória, mas em todos os momentos de minha vida.

A minha namorada, pelas palavras de incentivo e por estar ao meu lado sempre me motivando.

Ao professor Thiago Carvalho, pela sua orientação e seus ensinamentos, que tornaram possível a conclusão deste trabalho.

A todos os meus professores que me ensinaram o valor do conhecimento.

Aos amigos e colegas, por todos os momentos de alegria que me proporcionaram.

Agradeço também ao projeto ADVANCE por ter parcialmente financiado este trabalho.

EPÍGRAFE

"Para nós, os grandes homens não são aqueles que resolvem os problemas, mas aqueles que os descobrem". (Albert Schweitzer)

RESUMO

A Unified Modeling Language é um padrão de modelagem orientada a objetos bastante aceito e utilizado no desenvolvimento de sistemas. Entretanto, ela não garante a consistência de seus modelos por permitir múltiplas interpretações deles, o que causa inúmeros erros durante o desenvolvimento, que por sua vez impacta nos custos e prazos do projeto. Este trabalho tem como objetivo proporcionar um mecanismo para verificação do Diagrama de Casos de Uso proposto pelo Iconix através de uma ferramenta CASE que, além de proporcionar a criação e edição do diagrama, permita a sua transformação para a linguagem formal Event-B.

PALAVRAS-CHAVE: Event-B. Métodos Formais. Iconix. Diagrama de Casos de Uso. Desenvolvimento de Softwares. Verificação.

ABSTRACT

The Unified Modeling Language is a standard for object-oriented modeling widely accepted and used in systems development. However, it does not guarantee the consistency of their models, since it allows multiple interpretations of them, which causes numerous errors during the development, which in turn impacts on the project costs and deadlines. This work aims to provide a verification mechanism fot the Iconix Use Case Diagram through a CASE tool that, in addition to providing the creation and editing of this diagram, allows its transformation to the Event-B formal language.

KEYWORDS: Event-B. Formal Methods. Iconix. Use Case Diagram. Software Development. Verification.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

UML	Unified Modeling Language
NIST	National Institute of Standards and Technology
XP	Extreme Programming
RUP	Rational Unified Process
VDM	Vienna Development Method
CASE	Computer-Aided Software Engineering
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering
OMG	Object Management Group
RTF	Revision Task Force
FTF	Finalization Task Force
AMN	Abstract Machine Notation
GSL	Generalised Substitution Language
EMF	Eclipse Modeling Framework
XMI	XML Metadata Interchange
GEF	Graphical Editing Framework
MVC	Model, View, Controller
GMF	Graphical Modeling Framework
QVT	Query/View/Transformation
QVTR	QVT Relations
QVTC	QVT Core
QVTO	QVT Operations
MOF	Meta Object Facility
OCL	Object Constraint Language
CFG	Context-Free Grammar

SUMÁRIO

1 INTRODUÇÃO

O processo de desenvolvimento de um software consiste muitas vezes num conjunto de atividades agrupadas que possui os modelos parciais do sistema que será desenvolvido. Estes modelos abstraem o processo real do desenvolvimento e definem o ciclo de vida de um software que se inicia em sua concepção, passando pelo levantamento, análise e especificação de requisitos, projeto do sistema, codificação, testes, de acordo com cada funcionalidade ou como um todo, implantação e manutenção, sendo essas algumas das etapas primordiais para o desenvolvimento de um produto de qualidade. Para se alcançar todas as etapas que compreendem a produção de um sistema, os engenheiros de software utilizam-se de ferramentas e métodos diferentes que frequentemente ocasionam problemas de inconsistências.

A utilização de diagramas da UML (*Unified Modeling Language*)[1] é uma forma de facilitar o desenvolvimento, planejamento e manutenção de um sistema de software, sendo eles os mais utilizados e mais bem aceitos entre os analistas de softwares. Porém, apesar de ser uma abordagem bem madura, com seu constante crescimento e aprimoramento, que vem sendo feito desde a década de 1994, quando surgiu a UML, seus diagramas ainda possuem algumas ambiguidades, e são essas ambiguidades que podem prejudicar o andamento do projeto. Um dos problemas consiste em determinar se em algum momento uma alteração em um elemento de seus modelos acarretará em uma grande modificação nos modelos que o possuem como referência, seja ela visível graficamente ou não. Outro problema está em manter a semântica original do modelo de especificação mais abstrato após as mais variadas modificações e refinamentos que são realizados em seus modelos mais específicos. Mais claramente, deve-se verificar se as sucessivas modificações preservam a estrutura e o comportamento descritos no modelo de especificação.

Por não existirem mecanismos que façam a verificação dessas inconsistências já nas fases de especificação de requisitos, projeto e análise, que são fases iniciais do projeto, elas podem comprometer todas as regras de negócios do sistema, pois uma vez que um erro não seja detectado e tratado logo e acabe se estendendo por várias outras fases, o processo de refatoração dos estágios que as precedem demandará mais custo e impactará nos prazos do projeto. Uma pesquisa feita pelo NIST (*National Institute of Standards and*

Technology) em 2002, diz que a remodelagem de alguns sistemas ocasionou um consumo de 80% dos custo de desenvolvimento devido a problemas de relacionamento e inconsistências, reforçando que a existência de mecanismos capazes de detectar falhas e prover uma maneira mais rápida no seu tratamento, é algo imprescindível no processo de desenvolvimento.

Alguns processos de desenvolvimento adotam a verificação de inconsistências como um serviço obrigatório em suas fases. O Iconix[2], XP (*Extreme Programming*)[3], RUP (*Rational Unified Process*)[4] e Scrum[5] por exemplo, utilizam como técnicas: a inspeção dos modelos que é eficiente para demonstrar as funcionalidades, regras e etapas de um software, porém, por utilizar-se de linguagens imprecisas e visuais, sua verificação se torna extremamente dependente da capacidade de seus desenvolvedores e acaba se tornando bastante custosa; e a execução de teste, que se utiliza de ferramentas automatizadas, contudo, não abrange todas as verificações possíveis de um software, o que não garante a eliminação total dos problemas.

Os métodos formais, subárea da Engenharia de Software, se baseiam em conceitos matemáticos, lógica, estruturas matemáticas e teoria dos conjuntos, para especificação, desenvolvimento e verificação dos sistemas de software e hardware. Tais métodos servem como mecanismos para validar as decisões tomadas referentes ao projeto, e não para elaboração das mesmas. A utilização de modelos é a base de alguns métodos formais, como o B[6], Event-B[7],. VDM[8], Z[9], dentre outros, que lhes dão suporte a uma verificação mais precisa, rápida e automática das inconsistências encontradas. Porém, apesar de ser uma maneira que possa garantir a integridade do sistema, esses processos não são aplicados pelos desenvolvedores, pois uma grande parte deles não possuem capacitação para trabalhar com este tipo de abordagem, muitas vezes por não conseguir compreender a linguagem e os conceitos matemáticos envolvidos. Sendo assim, a junção de métodos formais com processos de desenvolvimento, aos quais os engenheiros de software e desenvolvedores já estejam habituados, permitiria, de forma transparente, que as validações fossem executadas sem que houvesse intervenção externa suscetíveis a erros.

A modelagem de software baseada na UML é bastante utilizada pelo mercado, porém, a verificação de sua integridade ainda não é muito usada. Apesar do aumento nas pesquisas relacionadas a este fator, poucas se concentram na consistência dos Casos de Uso. Alguns dos trabalhos, como [10], [11] e [12], estão focados na verificação dos casos

de uso. Porém, apenas o primeiro utiliza-se de métodos formais, onde, diferente deste, que usa o Event-B para proporcionar regras de validação, ele é usada a notação Z, uma notação em franco desuso. O segundo utiliza a linguagem declarativa OCL para expressar as restrições dos modelos da UML, o que aumenta a dificuldade na aplicação das restrições, uma vez que este tipo de linguagem é utilizado por pessoas com um vasto conhecimento matemático. O terceiro aborda um refinamento entre outros trabalhos, incluindo o [10] e o [13], que verifica a coerência entre os diagramas de classe, casos de uso, estados, atividade e sequência usando CFG (*Context-Free Grammar*)[14], que não fornece meios para a remoção das ambiguidades de suas definições.

Este trabalho tem como objetivo, fazer com que o Diagrama de Casos de Uso do Iconix, derivado da UML, possa ser mapeado para a linguagem formal Event-B, por meio de regras de transformação, proporcionando assim um mecanismo automatizado para verificação desse diagrama..

Iremos abordar a estrutura da UML em relação ao desenvolvimento, documentação e manutenção de softwares, e a possibilidade de usar uma linguagem formal em conjunto com seus diagramas, em especial o de Caso de Uso, como ferramenta auxiliar no processo de verificação. Iremos mostrar a elaboração de uma ferramenta CASE (*Computer-Aided Software Engineering*) que permita ao desenvolvedor criar um Diagrama de Casos de Uso e transformá-lo em uma especificação descrita na linguagem Event-B.

Esta monografia encontra-se organizada em 6 capítulos, incluindo esta introdução. No capítulo 2, iremos abordar a modelagem de sistemas, bem como descrever a UML, sua história, e seus diagramas e como a interpretação deles pode ser ambígua, além de mostrar um processo de desenvolvimento (Iconix) que se utiliza da UML e possui passos claros de verificação. Os métodos formais são utilizados para que se possa verificar a modelagem feita pela ferramenta CASE a ser desenvolvida, mais precisamente através do Event-B, que serão explicados no capítulo 3. As tecnologias utilizadas, para a estrutura física e visual, e a linguagem de transformação do diagrama de caso de uso, serão descritas no capítulo 4. Por fim, no capítulo 5, este trabalho irá explicar as regras que irão proporcionar a transformação do diagrama proposto em modelos Event-B. No capítulo 6 apresentaremos a conclusão deste trabalho, bem como, os trabalhos relacionados e futuros.

2 MODELAGEM

A modelagem de software é uma das principais atividades que levam ao desenvolvimento e implantação de um sistema de qualidade, pois é a partir dela que serão levantadas questões fundamentais que serão de suma importância para todas as fases de sua construção. Algumas dessas questões são os requisitos funcionais e os não funcionais. No primeiro, deve-se ter em mente o que o sistema deve ou não fazer, pensando em um produto que seja perfeito e que não tenha limitações de hardware e nem da plataforma que ele irá ser executado. No segundo, a modelagem também permite a criação de uma arquitetura que possibilite modificações visando um mínimo de desperdício de tempo, desenvolvimento e de custo. Ela nada mais é do que uma representação simplificada de algo real, em forma de modelos gráficos, que explica o comportamento e as características dos objetos que compõem um software, simplificando assim, o seu entendimento. Estes objetos servem para demonstrar as funcionalidades e as características que um sistema deve prover, por meio da análise de requisitos.

Os modelos gráficos simbolizam os relacionamentos e inter-relacionamentos existentes entre os objetos, permitem a visualização e o controle de sua arquitetura e melhora a compreensão do sistema. Cada modelo pode ser, por muitas vezes, melhor compreendido através da modelagem orientada a objetos, que é amplamente apresentada pela UML e onde o principal foco está na criação de classes e objetos, ou através da visão de um algoritmo, que foca na construção de funções e procedimentos, sendo essas duas apenas uma das diversas formas de se obter um modelo de organização. A seção a seguir é baseada no guia do usuário da UML[1], escrito pelos seus criadores.

2.1 Iconix

Criado por Doug Rosenberg, em 1993, o processo Iconix tem como objetivo juntar as melhores características das técnicas de metodologia que deram origem a UML (OMT[15], Objectory[16] e Booch[1]). Considerado um processo intermediário entre o processo RUP e o XP, ele apresenta uma metodologia poderosa no desenvolvimento de software. O Iconix utiliza um subconjunto, apenas quatro diagramas (casos de uso, classes, sequencia, robustez) dos 14 diagramas existentes atualmente na UML, como mostrado na

Figura 1.

Este processo conduz o desenvolvimento do software a partir dos requisitos. Suas várias etapas fazem com que eles sejam mais precisos e menos ambíguos, o que leva a construção de um software mais robusto. Segundo Rosenberg [17], cada um de seus diagramas compreende um propósito específico:

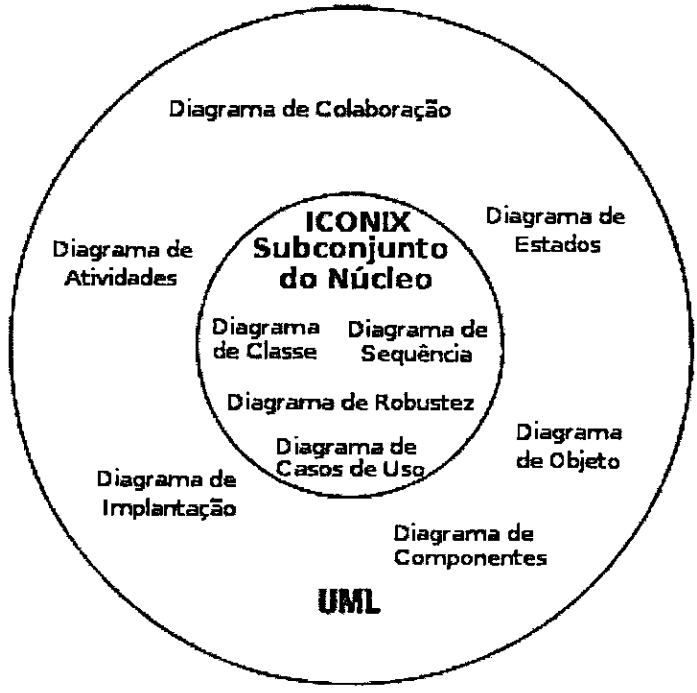


Figura 1: Processo Iconix – Subconjunto do núcleo da UML. Fonte: Adaptado de [17]

- Caso de uso: Define os requisitos comportamentais do sistema.
- Modelo de domínio/classe: Representação visual dos objetos do mundo real e de suas relações.
- Diagrama de Robustez: Reduz a ambiguidade dos requisitos comportamentais.
- Diagrama de Sequência: Define o comportamento dos objetos.

O Iconix se divide em dois eixos: dinâmico e estático. O primeiro consiste em representar os aspectos comportamentais de um software, já o segundo, representa os aspectos estruturais. Rosenberg [17] define, ainda, que esses eixos podem ser divididos em alguns passos, Figura 2, que são agrupados em quatro fases:

- Definição de Requisitos – definidos a partir do protótipo da interface

- Criação do modelo de domínio.
- Utilização dos casos de uso para definição dos requisitos comportamentais.

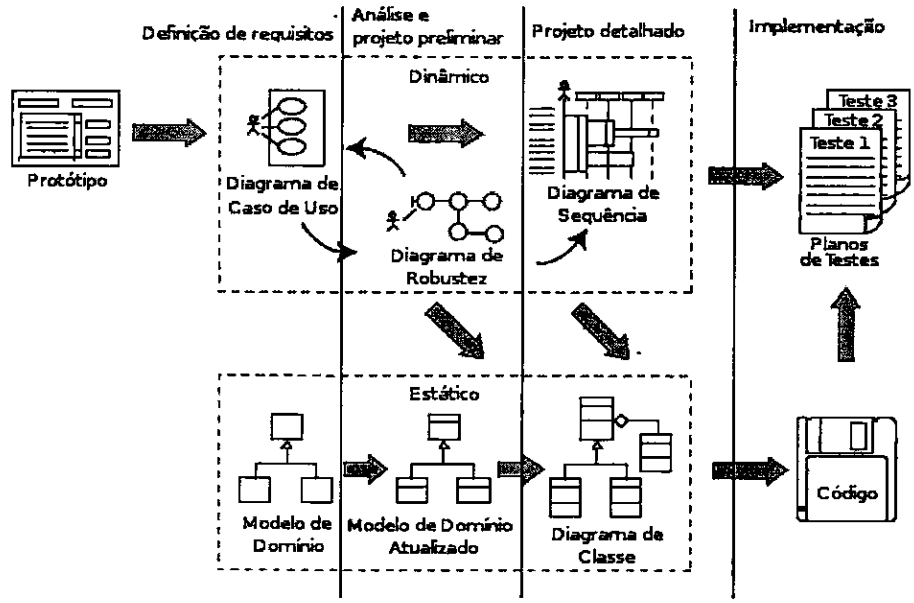


Figura 2: Processo Iconix. Fonte: Adaptado de [17]

- Análise e projeto preliminar
  - Realizar análise de robustez (diagrama de robustez), a fim de reduzir a ambiguidade os casos de uso.
  - Refinamento do modelo de domínio.
- Projeto detalhado
  - Modelagem do diagramas de sequência.
  - Refinamento do modelo de domínio (diagrama de classes).
- Implementação
  - Codificação.
  - Criação e execução do testes de aceitação do usuário.

2.1.1 Casos De Uso

Os casos de uso do processo Iconix se assemelham com os usados na UML, apresentando algumas diferenças com relação ao relacionamento existente entre os casos de uso e os atores. Os seus relacionamentos (generalização, *include* e *extend*) foram removidos, pois, para o Iconix, representa uma paralisia na análise ocasionada pelo tempo desperdiçado na decisão da utilização de algum deles. Ao invés disso, ele propõe que apenas dois esteriótipos sejam usados: o *invokes*, indicando que um caso de uso, ou um ator, invoca outro caso de uso; e o *precedes*, utilizado para indicar que um caso de uso é executado antes de outro. A Figura 3 apresenta a sintaxe do Event-B.

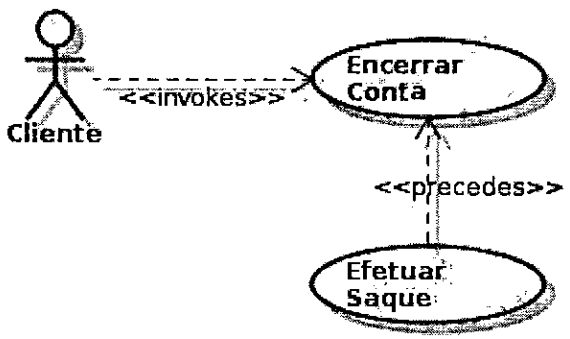


Figura 3: *Relacionamentos "invokes" e "precedes" no Iconix. Fonte: o autor*

2.2 UML

2.2.1 História da UML

Entre as décadas de 1970 e 1980, linguagens de modelagens orientada a objetos surgiram para explicar, utilizando-se de abordagens de design e análise, tanto as aplicações que estavam cada vez mais complexas, quanto as linguagens de programação orientadas a objetos que eram utilizadas por estas aplicações. Devido ao grande aumento na quantidade de métodos orientados a objeto, chegando a mais de 50 em apenas 5 anos, os usuários se depararam com métodos que não satisfaziam a necessidade de modelagem de seus sistemas, tendo que recorrer a vários deles, dando força a guerra dos métodos.

2.1.1 Casos De Uso

Os casos de uso do processo Iconix se assemelham com os usados na UML, apresentando algumas diferenças com relação ao relacionamento existente entre os casos de uso e os atores. Os seus relacionamentos (generalização, *include* e *extend*) foram removidos, pois, para o Iconix, representa uma paralisia na análise ocasionada pelo tempo desperdiçado na decisão da utilização de algum deles. Ao invés disso, ele propõe que apenas dois esteriótipos sejam usados: o *invokes*, indicando que um caso de uso, ou um ator, invoca outro caso de uso; e o *precedes*, utilizado para indicar que um caso de uso é executado antes de outro. A Figura 3 apresenta a sintaxe do Event-B.

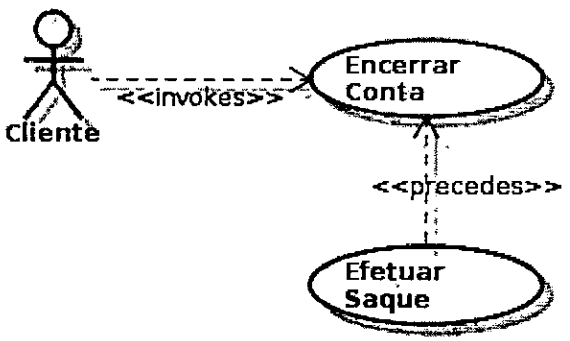


Figura 3: Relacionamentos "invokes" e "precedes" no Iconix. Fonte: o autor

2.2 UML

2.2.1 História da UML

Entre as décadas de 1970 e 1980, linguagens de modelagens orientada a objetos surgiram para explicar, utilizando-se de abordagens de design e análise, tanto as aplicações que estavam cada vez mais complexas, quanto as linguagens de programação orientadas a objetos que eram utilizadas por estas aplicações. Devido ao grande aumento na quantidade de métodos orientados a objeto, chegando a mais de 50 em apenas 5 anos, os usuários se depararam com métodos que não satisfaziam a necessidade de modelagem de seus sistemas, tendo que recorrer a vários deles, dando força a guerra dos métodos.

Percebendo estes problemas, alguns dos métodos criados posteriormente vieram com o intuito de acabar com esta guerra. Os métodos que se destacavam eram o Booch[1] de Grandy Booch, o modelo OMT (*Object Modeling Technique*)[15] de James Rumbaugh e o OOSE (*Object-Oriented Software Engineering*)[1] de Ivar Jacobson. O modelo Booch tinha maior expressividade nas fases de projetos e construção de projetos sugerindo como cada etapa poderia ser feita e sua abordagem era, inicialmente, através do diagrama de classe, além dos diagramas de estado, processo e interação. Em contra partida o OOSE se destacava pelo foco na análise, requisitos e *design* de alto nível utilizando-se de casos de uso para conduzir a modelagem, enquanto que o OMT era mais expressivo na análise de sistemas tratando dos dados provenientes da aplicação.

A fim de unir seus modelos, Booch e Rumbaugh se juntaram, em Outubro de 1994, na Rational Software Corporation, empresa na qual Booch trabalhava, dando início a criação da UML. Apesar de serem bastante semelhantes semanticamente, os dois modelos (Booch e OMT) possuíam suas diferenças principalmente quanto aos símbolos utilizados, necessitando serem unificados, e apesar de não serem métodos completos, pois ainda estavam sendo desenvolvidos, estes métodos eram tidos como líderes no processo de orientação a objetos. A UML 0.8 foi anunciada em Outubro de 1995, quando pouco tempo depois Jacobson se juntou a Rational e consequentemente ao projeto, expandindo seu escopo com o intuito de incorporar o modelo OOSE. No ano de 1996, com o lançamento da versão 0.9, muitas empresas já viam na UML um grande recurso para seus negócios. Empresas como a Digital Equipment Corporation, IBM, Microsoft, Oracle e a Rational juntaram recursos para tornar a UML mais forte e completa, criando, assim, uma linguagem bem definida, expressiva e que se adequava a uma grande variedade de problemas, a UML 1.0, versão esta que foi oferecida, em Janeiro de 1997, ao OMG (Object Management Group) em resposta a proposta de padronização da linguagem. Em Junho de 1997 foi oferecida a versão revisada 1.1 da UML ao OMG e aceita pela ADTF (Analisys and Design Task Force) em Setembro deste mesmo ano. A RTF (Revision Task Force) do OMG assumiu a manutenção da UML que sofreu algumas alterações que resultaram nas versões 1.3, 1.4 e 1.5. As alterações realizadas entre o ano de 2000 a 2003 resultaram na versão 2.0 da UML que foi revisada por cerca de um ano pela FTF (Finalization Task Force) da IBM, e a versão oficial só foi adotada no início de 2005 pelo OMG. Atualmente ela se encontra na versão 2.5, e possui um total de 14 diagramas.

2.2.2 Visão Geral

Através de modelos visuais o entendimento e a comunicação entre membros de um grupo de desenvolvimento faz com que eles tenham uma mesma ideia do sistema. É comum que empresas e desenvolvedores tenham seu próprio método de construção de software, porém, ao se utilizar destes métodos, pode ser que outros membros que venham a se integrar ao projeto tenham um certa dificuldade, além do tempo desprendido, para que se possa entender como esta nova metodologia irá funcionar e, ainda assim, não garante que a o entendimento correto do que foi especificado por já estarem acostumados a outros tipos de modelagem. A proposta da UML é a utilização de seus modelos gráficos e com semânticas bem definidas para que se possa acabar com os problemas descritos anteriormente.

A UML proporciona em sua linguagem modelos de especificação, que são amplamente utilizados pela análise, projeto e pela implantação de sistemas de softwares. Por possuir modelos que se adequam a diversas linguagens de programação orientadas a objeto como C++ e Java, a UML também é dita como uma linguagem para construção, apesar de seus modelos não definirem diretamente os códigos usados pelas linguagens, possibilitando, ainda, a visualização de relacionamentos existentes no banco de dados. A UML, ainda enquanto linguagem de construção, permite a criação de códigos a partir de seus modelos visuais e também possibilita a criação de modelos a partir de códigos, a chamada engenharia reversa. Entretanto, a engenharia reversa ainda precisa de intervenções externas para que se possa obter uma modelagem mais precisa, pois as ferramentas que produzem esta transformação não são capazes de expressar por completo o que está codificado.

Um outro aspecto abordado pela UML é a documentação de software. Esta documentação é, sem dúvidas, uma das principais atividades que devem ser realizadas na idealização, construção e manutenção de um sistema. Por meio desta, são explicados os funcionamentos físicos e lógicos, relacionamentos e limitações de funcionalidades que envolvem, direta ou indiretamente, todo o ciclo de vida de um software. A documentação é um aspecto primordial para a manutenção de qualquer sistema, onde, sem ela, a dificuldade na realização de alterações e melhorias em um sistema de grande porte seria extremamente difícil.

2.2.3 Diagramas

Os diagramas servem para representar diferentes aspectos e visões do sistema. Eles nos permitem visualizar os relacionamentos, inter-relacionamentos e como será feita a comunicação entre módulos ou até mesmo a comunicação entre o usuário e o sistema. Segundo a UML eles estão classificados em 2 tipos de diagramas, os estruturais e os comportamentais. A seguir, descreveremos os seus principais diagramas.

Os diagramas estruturais são usados na visualização, especificação, construção e documentação de aspectos estáticos de um sistema. Fazem parte desta nomenclatura cinco diagramas da UML, que são os diagramas de classe, objetos, componentes, implantação e pacotes.

- **O diagrama de classe** é o mais utilizado pelos usuários da UML e por ela própria, pois serve de base para outros diagramas. Nele são ilustradas as interfaces, classes, com seus atributos e métodos, e seus relacionamentos.
- **O diagrama de objeto** está diretamente relacionado ao diagrama de classes e é praticamente um complemento dele. Fornece um objeto, ou uma instância de uma classe, com os valores que podem ser armazenados em um determinado momento da execução do sistema.
- **O diagrama de componentes** contém os componentes de um software que podem ser reutilizados. Quando um componente é instanciado, as cópias de suas partes internas também são instanciadas [1]. Um exemplo bem claro disto está na forma que utilizamos as bibliotecas em nossos códigos, pois tais bibliotecas possuem um conjunto de classes que satisfazem uma necessidade específica. Outros componentes reconhecidos pela UML são: um arquivo executável, uma tabela no banco de dados, um documento ou um arquivo, contendo geralmente um código fonte.
- **O diagrama de implantação** é utilizado para determinar os elementos físicos que se comunicam com o sistema, um outro computador, um dispositivo, ou seja, os hardwares que podem se comunicar com o sistema. Este diagrama é utilizado em conjunto com o diagrama de componentes, pois um dispositivo pode conter um conjunto de componentes.
- **O diagrama de pacotes** é similar ao diagrama de componentes, mostra o

comportamento existente entre classes e suas colaborações.

Os diagramas comportamentais são aqueles que abordam o aspecto dinâmico de um sistema. Fazem parte desta categoria os diagramas de atividades, colaboração, sequência, transição de estados e casos de uso, que, por ser objeto de estudo deste trabalho, será descrito com mais detalhes na próxima sessão.

- **O diagrama de atividades** é como um gráfico de fluxo, ele descreve o fluxo de comunicação existente entre as atividades do sistema, mas ao contrário de um gráfico de fluxo comum, ele descreve as ramificações existentes no decorrer das escolhas realizadas por cada atividade.
- **O diagrama de sequência** se concentra na sequência em que os eventos acontecem, ou seja, os métodos e os objetos envolvidos em um determinado processo.
- **O diagrama de colaboração** assemelha-se ao diagrama de sequência. A diferença é que o segundo se concentra no fluxo temporal em que os eventos ocorrem. Já o primeiro se concentra na troca de mensagens e em como os objetos estão vinculados.
- **O diagrama de transição de estados** acompanha as mudanças que um objeto sofre dentro de um processo, podendo ser anexados a classes, casos de uso ou ao sistema inteiro.

2.2.4 Diagrama de Casos de Uso

Um caso de uso é uma descrição de um conjunto de sequências de ações que um sistema executa para produzir um resultado de valor observável por um ator do sistema[1]. O objetivo de um caso de uso é representar de forma geral e externa as funcionalidades e serviços que o sistema deverá prover a seus usuários.

Quando pensamos em construir algo, seja uma casa, um carro ou um software, devemos pensar nos aspectos que envolvem sua construção e o uso no qual ele se dará. Por exemplo, ao construir um sistema temos que expressar o comportamento do sistema na perspectiva do um usuário. Quando ele executa o sistema, ele deverá informar suas credenciais de acesso, para que posteriormente, devidamente autorizado, ele possa acessar todas as tarefas que a ele são destinadas. O ato de se identificar no sistema configura um

caso de uso, assim como, a interação de um usuário com as demais telas ou processos.

Os casos de uso podem descrever um sistema completo exibindo apenas os processos principais que ele deve realizar, porém, ao descrever um sistema, necessitamos saber um pouco mais sobre seu funcionamento. Por isso, eles podem ser divididos e podem ser precedidos ou até invocar outros casos de uso. Ao fazer isso estamos detalhando com mais precisão o passo-a-passo que deverá ser executado até que se obtenha o resultado final do processo que está sendo executado.

Todo caso de uso pode ser acessado por meio de um ator, representado por uma linha ligando ambos, ou até mesmo de um outro caso de uso, através de ligações como *include* e *extend*. Os relacionamentos de inclusão indicam uma obrigatoriedade, ou seja, a execução do primeiro ocasiona na execução obrigatória do segundo. Já o relacionamento de extensão, diz que um caso de uso pode ou não invocar outro, ou seja, quando há uma determinada situação em que uma tomada de decisão é necessária. Além disso existe um relacionamento entre os atores chamado de generalização, onde as características do ator alvo são incorporadas as do ator que originou o relacionamento.

3 MÉTODOS FORMAIS

Os métodos formais surgiram após as duas grandes crises do software ocorridas nas décadas de 60 e 70. A primeira crise ocorreu devido o surgimento de uma nova geração de computadores com uma maior capacidade, um aumento na dificuldade de resolução de problemas mais complexos e uma deficiência em tecnologia para o solucionar estes problemas. Os computadores passaram a se tornar não só uma máquina para resolver cálculos, mas uma máquina capaz de efetuar tratamentos e processamentos de dados cada vez maiores. A segunda crise surge com o foco no maior custo do software com relação ao hardware, onde, naquela época, o empenho maior estava na construção de sistemas e a relevância do software era pouca.

Pelo surgimento de sistema cada vez mais complexos e robustos, a garantia da qualidade dos software era cada vez mais difícil de se alcançar. A utilização de métodos informais para sua verificação tornou-se cada vez mais obsoleta, imprecisa, incompleta e ineficiente. Já os métodos formais são mais precisos, suas técnicas se baseiam na matemática, na lógica e na teoria dos conjuntos para descrever um modelo matemático de um sistema, resultando na diminuição de erros.

Outra vantagem que se pode apontar é que, por possuir uma semântica bem definida, possibilita uma melhor percepção dos requisitos do software, assim como, a prova de que os requisitos iniciais compreendem o sistema desenvolvido, reduzindo o custo final. Todavia, existem ainda algumas desvantagens no uso de métodos formais como, por exemplo, a falta de ferramentas que trabalhem com estes métodos e a dificuldade em mostrar para os clientes que, apesar do alto custo inicial, sua utilização diminuirá o custo total do sistema.

Os métodos formais são usados para especificação, desenvolvimento e verificação de sistemas, além de buscar uma maior qualidade de software através de refinamentos, testes e provas formais. Para tanto, eles se utilizam de linguagens de especificação, que consiste em notações matemáticas com semântica, vocabulário e sintaxe precisamente bem definidas, e são usados para verificar se um sistema condiz com sua especificação funcional, o que permite a detecção de problemas antes do seu desenvolvimento. Estas linguagens são divididas em dois tipos: orientadas a modelo e baseadas em propriedades (axiomáticas e algébricas).

- **Linguagem orientada a modelo:** utilizam modelos matemáticos . O sistema é baseado em um modelo de estados que representam algo do mundo real, e utilizam os conceitos matemáticos de conjuntos, listas e operações sobre os estados, para o seu desenvolvimento. Algumas linguagens orientadas a modelo são: Z, VDM e B.
- **Linguagens baseadas em propriedades:** ao contrário das linguagens orientadas a modelos, as baseadas em propriedades utilizam um conjunto de estados para representar as propriedades externas de um sistema, ou seja, o que ele deve fazer. Esta linguagem se divide em dois subgrupos, axiomáticas e algébricas.
  - **Axiomáticas:** Este subgrupo usa a abstração do software baseada em conceitos lógicos. A linguagem Anna (Annoted Ada)[18], uma extensão da linguagem Ada, é um exemplo de linguagem de especificação axiomática.
  - **Algébricas:** Como uma forma de simplificar as notações complexas utilizadas nos métodos formais, as linguagens algébricas começaram a ser utilizadas, pois permitem que as equações sejam mais simples e acessíveis, por meio de equações algébricas. As entidades e objetos, semelhantes as classes e tipos, utilizadas são denominadas de *sort*, que nada mais é de que um conjunto de objetos. Ela é dividida em quatro módulos: introdução: onde se define a entidade (representação de algo real, com sua descrição), *sort* (uma classe) e o nome das aplicações necessárias (pacotes ou classes utilizadas no contexto, os *imports*); a descrição informal, que descreve o que cada método faz; a assinatura, utiliza um conjunto de operações (*constructor* e *inspection*) que definem as características e propriedades de um objeto. O *constructor* contém três outras operações (*create*, *insert* e *remove*) relacionadas a criação e remoção de objetos, e o *inspection* outras duas (*last* e *eval*) que validam as operações; e os axiomas, que consistem em um conjunto de equações que se baseiam em expressões algébricas.

A especificação formal serve como base para o desenvolvimento e verificação de sistemas de software, pois ela descreve os detalhes que o contemplam, o que garante que a codificação e a execução dos testes, que irão validar e verificar sua integridade, será realizada de forma precisa e correta. Além de se basear em cálculos e elementos matemáticos, os métodos formais tem como base outros três conceitos fundamentais:

invariantes, estados e operações.

- **Invariantes:** As invariantes são propriedades que não podem ser alteradas devido a suas restrições. Elas são amplamente encontradas na matemática, principalmente na transformação de alguns números em outros, onde a necessidade de encontrar um padrão, uma constante, torna-se necessário, e é a essa constante que denomina-se invariante.
- **Estados:** As condições que um determinado objeto pode assumir em sua existência são denominados de estados. Estes estados são limitadas ao escopo do objeto e das condições que a ele foi atribuída, ou seja, ele pode assumir apenas um conjunto de estados finitos.
- **Operações:** Contemplam a função de verificação dos parâmetros dos métodos, antes e depois de sua execução, se os parâmetros foram enviados corretamente e se a execução do método foi bem sucedida.

3.1 Método B

Desenvolvido por Jean-Raymond Abrial na França e Reino Unido, em 1980, o método B[6], como sitado anteriormente, também é um método de especificação formal, e como tal, garante que as especificações iniciais do sistema sejam rigorosamente atendidas, garantindo que suas propriedades sejam coerentes, não ambíguas e a não contraditórias. Largamente é utilizado em sistemas críticos, como sistema ferroviário, aeroespacial ou simplesmente na área de pesquisa e desenvolvimento.

Ele utiliza a linguagem B como base para o desenvolvimento de softwares, trabalha em mais baixo nível quando comparado ao método Z, também desenvolvido por Abrial, e suas ferramentas possuem como base a AMN (*Abstract Machine Notation*)[6], o que proporciona uma melhor e mais fácil aplicabilidade de uma especificação do que o Z. Este método tem como principais componentes a teoria dos grupos (máquinas abstratas) e a lógica de primeira ordem. Nas próximas subseções apresentaremos os componentes dessa linguagem.

3.1.1 Máquinas Abstratas

As máquinas abstratas mantêm o estado e um conjunto de operações do programa. Elas proveem operações que possibilitam efetuar alterações nos estados. O significado de suas operações é dado pela semântica GSL (*Generalised Substitution Language*). Segundo seu desenvolvedor, Abrial, GSL é uma notação para atribuir significado a programas, onde um programa é entendido como um "predicado transformador", que relaciona os estados que o antecedem e o precedem, e funcionam como expressões matemáticas.

```
MACHINE <nome da máquina>
VARIABLES
        <nome da variável>
        ...
INVARIANT
        <nome da invariante>
        ...
INITIALIZATION
        <nome da variável> : <valor inicial>
        ...
OPERATIONS
        <operação>
        ...
END
```

Figura 4: Máquina abstrata. Fonte: o autor

Na Figura 4, a cláusula **MACHINE** corresponde ao nome da máquina, que também pode trazer parâmetros declarados. A cláusula **VARIABLES** contém um conjunto de variáveis que representam o estado da máquina. Na cláusula **INVARIANT** são adicionadas regras que são obedecidas por todas as operações na máquina. Na cláusula **INITIALIZATION** é onde a máquina é inicializada, ou seja, onde as variáveis recebem seus valores iniciais. A cláusula **OPERATIONS** que contém uma lista das operações utilizadas para alterar os estados da máquina. Por fim a cláusula **END** indica o fim da especificação da **MACHINE**.

3.1.2 Refinamento

Os refinamentos ocorrem sempre que uma máquina precisa ser mais detalhada. Quando modelamos um software, inicialmente temos a falsa ideia de abstrairmos todo o sistema, suas funcionalidades, seus requisitos, seus problemas e seus alvos principais. Mas, no decorrer da análise, vemos a necessidade de dividi-lo em módulos para expressar melhor cada nível de sua estrutura. A esses módulos precisamos ainda avançar cada vez mais em nossa abstração até conseguirmos chegar a algo mais concreto, algo que seja facilmente traduzido para uma linguagem de programação de alto nível. Todos estes processos de detalhamento que utilizamos, podem ser chamados de refinamentos. A Figura 5 mostra a especificação de um refinamento que é semelhante a uma *Machine*, apresentando sutis diferenças. A cláusula **REFINEMENT** correnspponde ao nome do refinamento, e a cláusula **REFINES** contém o nome da máquina a ser refinada.

REFINEMENT <nome do refinamento>
REFINES <nome da máquina a ser refinada>
VARIABLES
    <nome da variável>
    ...
INVARIANT
    <nome da invariante>
    ...
INITIALIZATION
    <nome da variável> : <valor inicial>
    ...
OPERATIONS
    <operação>
    ...
END

Figura 5: Refinamento. Fonte: o autor

Esta analogia nos permite perceber que, quando refinamos um modelo, nós não estamos compactando sua estrutura, não vamos necessariamente remover partes dele para que fique cada vez menor. Na maioria das vezes, o conjunto das variáveis iniciais aumenta a medida que outros refinamentos acontecem. Isto acontece porque entramos em um nível maior de complexidade. Um problema encontrado no refinamento é que ele não permite que novas operações sejam acrescentadas. Por fim, após passar pelos refinamentos

necessários, alcança-se uma versão final, denominada Implementação que pode ser traduzida para linguagens de programação como Ada, C ou C++.

3.1.3 Implementação

Uma implementação (*implementation*) inicia com a cláusula **IMPLEMENTATION**, onde está o nome da estrutura, e da cláusula **REFINES**, que indica qual o refinamento que o implementa. Ela não tem um estado próprio, ou seja, sozinha ela não implementa nenhuma especificação. Para modelar o estado de seu refinamento, ela "importa" outras especificações.

No decorrer dos refinamentos de uma máquina abstrata, suas operações podem ser entendidas como as interfaces (*interfaces*) da linguagem de programação orientadas a objeto, por possuir somente a assinatura dos métodos e somente as classes que a implementam possuem de fato a implementação de seus métodos. Todo este processo de refinamento e implementação garante que o código final prove que é consistente com a especificação inicial.

3.3 Event-B

Desenvolvido por Abrial [7], mesmo criador do método B, o Event-B é um método formal que desempenha suas funções a nível de modelagem. Atualmente ele é utilizado na plataforma open source Rodin, um IDE baseada no Eclipse, que oferece meios para refinamento e provas matemáticas. Esta plataforma recebe apoio pelo projeto Advance (2011 a 2014), tendo sido financiado anteriormente pela Deploy (2008), ambos projetos da União Europeia, e RODIN (2004 a 2007).

Considerado uma evolução do método B, descrito na seção anterior, esta linguagem tem como principal característica o uso da teoria dos conjuntos para modelagem, refinamentos, que representam os diferentes níveis de abstração do sistema, e o uso de provas matemáticas para verificar a veracidade dos refinamentos e propriedades do sistema. As seções que descrevem a estrutura desta linguagem são baseadas no manual do Event-B.[19].

3.3.1 Máquinas e Contextos

Máquinas e contextos são derivações do conceito de modelos utilizados no Event-B que, segundo Abrial, contém o desenvolvimento matemático completo de um Sistema de Transição Discreto. As máquinas podem ser refinadas (*Refines*), em uma máquina concreta, para se obter uma precisão maior de suas características, que a cada refinamento outras podem ser atribuídas ou removidas de sua estrutura, e de suas complexidades. Os contextos podem estender outros contextos (*Extends*) e uma máquina pode ver (*sees*) um ou muitos contextos, como mostrado ma Figura 6.

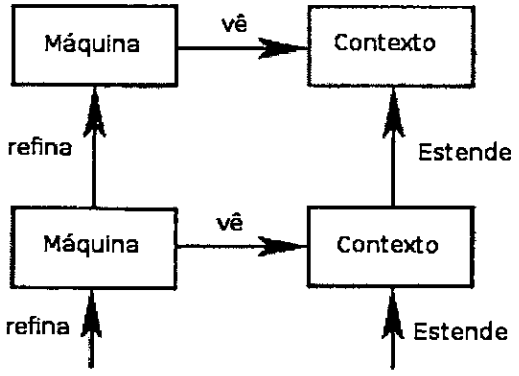


***Figura 6:** Relação entre máquinas e contextos. Fonte: Adaptado de* [20]

As máquinas do Event-B, componentes obrigatórios na modelagem, descrevem o comportamento dinâmico de um modelo por meio de variáveis cujos valores são alterados pelos eventos[19]. Elas são compostas pelas seguintes cláusulas: *MACHINE, REFINES, SEES, VARIABLES, INVARIANTS, EVENTS, VARIANTS* e *END*. As cláusulas que possuem um * não são obrigatórias em uma máquina.

- **Machine:** representa o nome da máquina.
- **Refines** *: indica que existe um refinamento em uma máquina mais abstrata, contendo o nome dela.
- **Sees** *: esta cláusula possui o relacionamento com um contexto.
- **Variables:** especifica uma lista distinta de variáveis de estado que representam o estado da máquina.
- **Invariants:** as invariantes, implicam regras, ou leis, que os estados devem cumprir.
- **Events:** conjunto de eventos, ou transições. Os eventos são responsáveis por alterar

os estados da máquina. Eles são divididos da seguinte forma: *EVENT, REFINES, ANY, WHERE, WITH THEN, END.* Porém, existe um evento em especial que não partilha desta divisão, ele é denominado *INITIALIZATION*, sendo nele que todas as variáveis são inicializadas.

  - **Event:** representa o nome do evento.
  - **Refines *:** especifica qual evento abstrato está sendo refinado.
  - **Any *:** representa os parâmetros de um evento, se existir.
  - **Where *:** corresponde às guardas de um evento, uma lista de predicados. Uma guarda ou *guard*, representa as condições para que um determinado evento ocorra, todavia, eles não são aplicados ao evento *INITIALIZATION*, pois não existem pré-condições para a inicialização das variáveis.
  - **With *:** esta cláusula contém as testemunhas. Quando um evento concreto refina um abstrato que é parametrizado, então todos os parâmetros abstratos deve receber um valor, chamado de testemunha, no evento concreto.
  - **Then:** nesta cláusula são definidas as ações (*actions*) que realizam as alterações que as variáveis irão sofrer sempre que as condições (*guards*) sejam satisfeitas. As variáveis que não estão no escopo das ações não são afetadas.
- **Variant *:** contém uma expressão numérica que é decrementada sempre que um evento ocorre, para passar a vez a outro.
- **End:** indica o o fim da especificação da máquina.

A Figura 7 mostra a sintaxe de uma máquina e um evento, respectivamente.

Os contextos são um caso particular de modelos, eles representam a estrutura estática do sistema. Eles possuem a seguinte divisão: *CONTEXT, EXTENDS, SETS, CONSTANTS, AXIOMS* E *EXTENDS.*As cláusulas que possuem um * não são obrigatórias em um contexto.

- **Context:** representa o nome do contexto. Não pode possuir o mesmo nome de uma máquina.
- **Extends *:** utilizada quando um contexto estende um ou muitos contextos, podendo adicionar outros elementos a sua estrutura e pode usar todas as constantes e axiomas do contexto estendido.

- **Sets *:** conjunto globais referenciados pelas máquinas.
- **Constants *:** seção onde as constantes são declaradas e podem ser utilizadas por axiomas ou máquinas.

a)
```
MACHINE <nome da máquina>
REFINES <nome da modelo refinado>
SEES
        <nome do contexto>
        ...
VARIABLES
        <nome da variável>
        ...
INVARIANTS
        etiqueta : predicado
EVENTS
        <evento>
        ...
VARIANT
        <variante>
        ...
END
```

b)
```
EVENT <nome do evento>
        REFINES
                <nome do evento a ser refinado>
                ...
        ANY
                <nome do parâmetro>
                ...
        WHERE
                etiqueta : predicado
        WITH
                etiqueta : testemunha
        THEN
                etiqueta : ação
END
```

Figura 7: Sintaxe de uma máquina(a) e um evento (b) Event-B. Fonte: o autor

Axioms *: contém uma lista de regras que serão aplicadas aos elementos do contexto.

- **End:** indica o fim da especificação do contexto.

A Figura 8 a seguir, mostra a sintaxe de um contexto.

```
CONTEXT <nome do contexto>
EXTENDS <nome do contexto estendido>
SETS
        <nome do conjunto>
        ...
CONSTANTS
        <nome da constante>
        ...
AXIOMS
        etiqueta : predicado
        ...
END
```

Figura 8: Sintaxe de um contexto Event-B. Fonte: o autor

3.4 Ferramentas de Suporte

Atualmente existem algumas ferramentas que trabalham com o método B. O B-Toolkit[21], consiste em um conjunto de ferramentas que estão totalmente integradas para agilizar o desenvolvimento de softwares. O B-Toolkit usa uma interface X Windows Motif (interface gráfica utilizada para sistemas derivados do UNIX) que permite o acesso a estas ferramentas. Outro método formal considerado uma evolução do método B, mais simples e de fácil compreensão, tem sido desenvolvido nos últimos anos, chamado de Event-B que possui como suporte uma ferramenta chamada Rodin (Rigorous Open Development Environment for Complex System).

O Rodin[22] foi criado, em 2004, como uma ferramenta de código-livre, tendo como base o Eclipse. Ele foi estendido pelo projeto DEPLOY[23], em 2007, que busca incorporar melhorias para esta plataforma e provê suporte até os dias atuais. Atualmente ele se encontra na versão 2.8. Ele fornece um excelente mecanismo para análise sintática, refinamento e provas matemáticas, além de possuir suporte para verificação e animação de modelos, suporte para recursos de interfaces gráficas e pode ser facilmente estendido por meio de plugins.

A plataforma Rodin é constituída basicamente por cinco camadas, são elas:

- Núcleo do Rodin: É uma extensão do Eclipse, e é formado por dois componentes do Java Development Tools of Eclipse:
  - O repositório: que armazena os elementos de dados, de forma persistente, em forma de objetos Java, em um arquivo XML.
  - O construtor: gerencia as alterações realizadas nos elementos do repositório.
- Biblioteca do Event-B: onde reside o analisador sintático e o mecanismo de provas matemáticas.
- Núcleo do Event-B: é composto por três módulos:
  - Analisador estático
  - Gerador de obrigação de provas
  - Gerenciador de obrigação de provas
- Interface gráfica do Event-B: representa a parte gráfica da plataforma, semelhante a IDE Eclipse, e se divide em duas interfaces:
  - Interface de modelagem e interface para criação e verificação das regras

4 TECNOLOGIAS

Neste capítulo iremos mostrar as tecnologias utilizadas para o desenvolvimento de uma ferramenta CASE com o objetivo de modelar, além de possibilitar a verificação formal do diagrama de caso de uso do Iconix. Para tal, iremos desenvolver um plugin que será executado na plataforma Rodin, derivada do Eclipse, que já possibilita a verificação dos modelos do Event-B.

4.1 EMF

O EMF (Eclipse Modeling Framework)[24] é um framework de modelagem, desenvolvido para Eclipse, que permite que desenvolvedores criem rapidamente ferramentas e outras aplicações robustas baseadas em modelos simples, também chamados de meta-modelos. Estes meta-modelos podem ser representados por qualquer diagrama. A notação padrão usada para descrever esses modelos é a UML que pode ser usada como uma entrada, a partir da qual uma aplicação é gerada. Por padrão ele utiliza o XMI (XML Metadata Interchange) para persistir a definição do modelo e pode ser utilizado por outras ferramentas e aplicações. Pode-se criar um meta-modelo através de um editor XML, a partir de um aplicativo de modelagem ou usando anotações Java, por exemplo. [24]

Uma vez especificado o meta-modelo, a partir dele pode-se gerar as implementações correspondentes a classes em Java, permitindo a geração de códigos corretos, eficientes e facilmente customizáveis. Um meta-modelo é baseado em outros dois meta-modelos, o Ecore e o Genmodel. O Ecore é a estrutura de modelo mais essencial, pois ele representa todas as informações que um modelo contém para a definição das classes. O Genmodel (.genmodel) fomece as informações específicas da plataforma, e é necessário para a geração do código. No EMF podemos modelar nosso meta-modelos de duas formas. A primeira é utilizando o Ecore Model (.ecore), um arquivo de nos permite visualizar e modelar de forma textual. A segunda é por meio do Ecore Diagram (.ecorediag), que só pode ser gerado se existir o modelo .ecore, onde podemos modelar de forma gráfica os nosso elementos, ou seja, eles são expressos usando a notação UML. O Ecore define quatro tipos de elementos, são eles:

- EClass: representa as classes, que podem conter qualquer quantidade de

super-classes, referências (agregação ou herança) e atributos.

- EAttribute: definem os atributos de uma Eclass, representados pelo nome e tipo do atributo, separados por ":".
- EReference: representa uma associação entre as classes, podendo ser uma relação de agregação ou herança.
- EDataType: representa o tipo de um EAttribute, assumindo os tipos de dados utilizados na linguagem Java.

Uma outra framework a ser citada é o GEF (Graphical Editing Framework) [25]. Ele é um framework para criação de editores gráficos que segue o conceito de MVC (Model, View, Controller), ou seja, ele separa o modelo (Model), a sua visão (View) e a lógica do programa (Controller). Os seus modelos contém a estrutura do programa e como os dados serão armazenados. Toda a representação gráfica existente é feita através do Draw2D[26], que é um kit de ferramentas de layout e renderização construído em cima do SWT. Pode ser usado independente (fora do Eclipse) ou em combinação com o GEF. Por fim, o *controller* é responsável por permitir a comunicação entre a visão e os modelos.

4.2 GMF

Segundo a wiki oficial [27], o GMF (Graphical Modeling Framework) "fornece um componente gerador e infra-estrutura em tempo de execução para o desenvolvimento de editores gráficos baseados em EMF e GEF". Servindo como uma espécie de ponte, ele permite que utilizemos as duas frameworks (EMF, GEF) de uma forma mais confortável, ao invés de trabalharmos com ambos separadamente.

Segundo [28], o ambiente de execução utiliza-se de alguns componentes do EMF como apoio *clipboard*, que fornece aos usuários facilidade em copiar e colar os EObjects, e a extensão dos tipos de elementos, além dos elementos do GEF com adição de novos elementos gráficos. A *framework* de generalização contém editores especiais para lidar com os modelos GMF e um gerador que produz o editor de código a partir dos modelos GMF. Ele usa quatro modelos diferentes – três (.gmfgraph, .gmftool, . gmfmap) para criar o mapeamento entre um modelo Ecore (EMF) e sua representação e um (.gmfgen) para gerar o código de editor visual (GEF). Na Figura 9, vemos a estrutura de

um projeto GMF sem a utilização do Ecore. Os modelos do GMF são listados a seguir:

- Modelo de definição gráfica – Este modelo (.gmfgraph) é responsável por definir as formas e figuras, que irão representar meta-modelos Ecore criados pelo EMF, e os controladores, que são responsáveis por gerenciar toda a interface gráfica dos modelos.
- Modelo de definição de ferramentas – Compreende um conjunto de ferramentas (.gmftool), atalhos, que serão utilizadas para criar os modelos gráficos que foram definidos no .gmfgraph, ou seja, a partir dele selecionaremos os elementos para criar nosso diagrama.
- Modelo de mapeamento – Por meio deste modelo (.gmfmap), os meta-modelos são mapeados para seus respectivos controladores e elementos gráficos. Ele mapeia os elementos criados no .ecore juntamente com os elementos criados no .gmfgraph e .gmftool.
- Modelo de geração de código – Através dele (.gmfgen) são gerados os códigos para o editor gráfico. Ele é considerado um arquivo de configuração pois contém todas as definições de criadas nos arquivos anteriores.

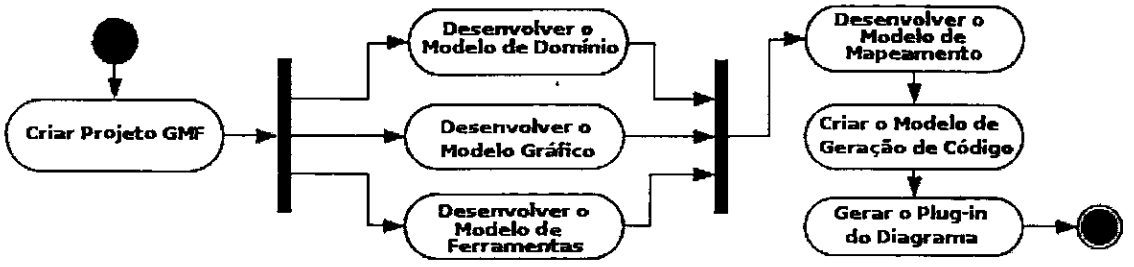


Figura 9: Estrutura de um projeto GMF. Fonte: Adaptado de [29]

As tecnologias vista até o momento servem para definir a estrutura física e visual dos componentes necessários para a criação da ferramenta CASE proposta neste trabalho, entretanto, uma outra é necessária para que possamos transformar estes elementos em modelos Event-B, o modelo QVT. A seção a seguir é baseada no documento de especificação formal da OMG. [30]

4.3 QVT

Criado pelo OMG, em 24 de Abril de 2008, com o intuito de padronizar as transformações realizadas de modelo para modelo, o QVT (Query/View/Transformation) foi proposto para tratar da manipulação dos modelos de outra de suas tecnologias, o MOF 2.0 (Meta Object Facility). Os conceitos do QVT representam:

- Query: A consulta consiste em uma expressão que um modelo avalia, onde são retornados uma ou várias instâncias do modelo alvo. A OCL (Object Constraint Language) é um exemplo de uma linguagem de consulta, e é utilizada pelo QVT.
- View: A visão está diretamente ligada a um modelo. Ela não pode ser alterada, a menos que o modelo seja também alterado.
- Transformation: Um transformação gera, para cada modelo fonte, um modelo alvo. Ela pode ser independente, onde não existe relação entre os modelos, ou dependente, quando ambos os modelos estão relacionados, ainda podendo ser unidirecional, para cada alteração no modelo fonte o alvo deve ser alterado, porém o inverso não é permitido, ou bidirecional, permitindo que uma alteração em um dos modelos afete o outro.

Esta especificação possui uma arquitetura híbrida composta por uma parte declarativa e imperativa. A primeira possui dois níveis (*Relations* e *Core*), onde o processamento da transformação é executado, necessitando de um compilador e um interpretador para executar a linguagem, e duas implementações imperativas (*Operational Mappings* e *Black Box*). Este trabalho aborda a arquitetura imperativa, discorrendo, mais especificamente sobre, o Operational Mappings. A Figura 10 mostra a relação entre os meta-modelos do QVT.

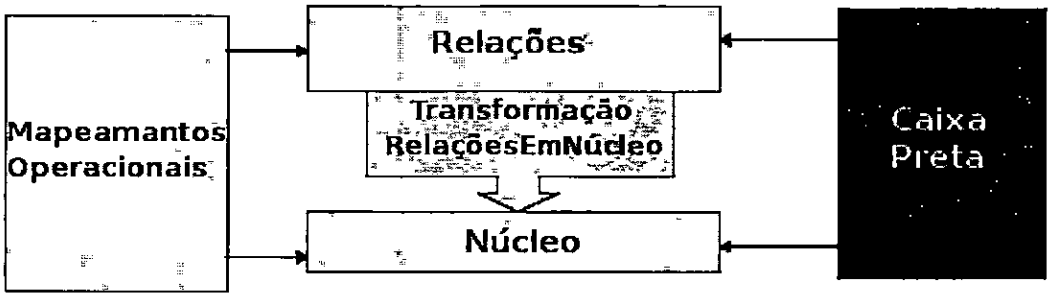


Figura 10: Relação entre os meta-modelos QVT. Fonte: Adaptado de [30]

- Relations: Também conhecida como (QVTR) É uma linguagem que possui uma sintaxe gráfica concreta, suporta objetos complexos, e cria, implicitamente, classes e instâncias dos registros processados na transformação.
- Core: Esta linguagem/modelo (QVTC) suporta apenas padrões que avaliam as variáveis contra um conjunto de modelos. Ele é tão poderoso quanto o Relations, porém, as transformações que o utilizam são mais detalhadas, o que é possível graças a sua semântica mais simplificada. Também pode ser usado para referência pelo *Relations*.
- Operational Mappings: Ou simplesmente QVTO, é uma extensão das camadas *Relations* e *Core*. É unidirecional, ou seja, existe apenas um modelo base, podendo existir um ou muitos modelos alvos. Possui características das linguagens imperativas, como loops e condições, além de possuir sintaxe parecida com a mesma.
- Black Box: Permite o uso de bibliotecas que não são nativas do QVT e integram facilmente outras linguagens.

4.3.1 QVTO

O QVT *Operational Mapping* permite que sejam definidas transformações operacionais ou imperativas, graças a sua abordagem híbrida. As transformações operacionais representam uma transformação unidirecional, ou seja, ela transforma um modelo de origem em um modelo alvo, onde quaisquer alterações realizadas no modelo alvo não remete alterações no modelo origem. Ademais, elas definem a assinatura de seus mapeamentos, indicando quais os modelos relacionados a ele, e deve possuir uma única operação principal (denominada *main*), que define, explicitamente as etapas de execução das transformações.

Os modelos que devem ser enviados as operações possuem tipos. Esses tipos definem um conjunto de pacotes MOF (metamodelos), um conjunto de condições e uma conformidade, estrita ou efetiva. O exemplo a seguir mostra uma assinatura (*usecase2eventb*), que transforma um caso de uso em modelos Event-B, seu modelo de entrada (*inoutModel*) e o tipo do modelo definido como *eventb*.

A figura 11 mostra um tipo de modelo (*modeltype*) chamado eventb que usa os meta-modelos do ecore do Event-B, o nome de uma transformação, chamada usecase2eventb, e a assinatura de um mapeamento, chamado iuml2eventb.

```
modeltype eventb uses core('http://emf.eventb.org/models/core');

transformation usecase2eventb(inout inoutModel:eventb);
main() {
  inoutModel.rootObjects()[eventb::machine::Machine]->map iumlb2eventb();
}
```

Figura 11: Definição de uma assinatura QVTO. Fonte: o autor

Assim como as linguagens imperativas, como Java, o QVTO permite que suas operações e tipos sejam facilmente reutilizados, por meio de bibliotecas (*library*). Uma biblioteca pode ser estendida (*extends*) pelas transformações, ou seja, todas as operações definidas nela se comportam igualmente as que foram definidas na transformação.

As operações de mapeamento (*map*) implementam os mapeamentos entre os modelos origem e os modelos alvos. Elas podem conter uma guarda (cláusula *when*), o corpo do mapeamento e uma pós-condição (cláusula *where*), e são refinamentos dos modelos a que elas são atribuídas. Além disso, elas podem preceder uma disjunção (*disjuncts*), que é uma conjunto de outras operações de mapeamento, onde o resultado das cláusulas *when* de suas disjunções deve ser satisfeito para que o retorno seja o esperado, caso contrário o valor um valor nulo (*null*) será retornado. A Figura 12 mostra um exemplo do uso de alguns elementos citados anteriormente.

```
import usecase2variables;

transformation usecase2eventb(in usecasein : usecasediagram, out eventbout : eventb)
   extends usecase2variables;

main() {
    inoutModel.rootObjects()[eventb::machine::Machine]->map iumlb2eventb();
}

mapping iumlb2eventb::OperationMapping() : eventb::machine::Machine
   disjuncts iuml2eventb::Disjunct1, iuml2eventb::Disjunct2 {}

mapping iuml2eventb::Disjunct1() : eventb::machine::Machine
    when {self.name <> ""}{
       name := self.name;
    }

mapping iuml2eventb::Disjunct2() : eventb::machine::Machine
    when {self.name <> null}{
       name := self.name;
    }
```

Figura 12: Operações de mapeamentos, importação e disjunções. Fonte: o autor

5 O PLUGIN USECASE2EVENTB

Neste capítulo, iremos apresentar o processo para a criação e desenvolvimento de uma ferramenta CASE, a estrutura do Diagrama de Casos de Uso utilizado no Iconix e as regras propostas para a transformação dos elementos do nosso diagrama.

No capítulo 5, abordamos as tecnologias que foram utilizadas para o desenvolvimento e criação da nossa ferramenta, assim como a transformação dos meta-modelos. Como já dissemos, ele será utilizado como um plugin juntamente com a plataforma Rodin, derivada do Eclipse, que oferece suporte para a verificação dos artefatos Event-B. Utilizamos o GMF para modelar os meta-modelos, fazer seus mapeamentos e criar o menu de acesso aos elementos do diagrama, o que possibilita a criação e edição de diagramas de casos de uso. Além disso, faremos uso do QVT para realizar a transformação dos casos de uso em Event-B.

5.1 Especificação dos Meta-modelos

O meta-modelo proposto não segue a mesma especificação do OMG, utilizada na UML. Ele contém apenas os elementos principais (atores, casos de uso e o relacionamento entre eles), possuindo uma especificação mais simplificada que é utilizada no Iconix.

Para manter a conformidade com o Iconix, que defende apenas o uso de dois esteriótipos que relacionam os atores e os casos de uso, duas meta-classes foram incluídas, a *BIconixPrecedes* e a *BIconixInvokes*.

O prefixo "BIconix" foi adicionado a todas as meta-classes com o intuito de evitar confusão com relação ao diagrama do OMG. Na Figura 13 podemos visualizar o meta-modelo do diagrama de casos de uso. Nas subseções seguintes, iremos descrever com mais detalhes todas as meta-classes modeladas segundo a especificação do processo Iconix.

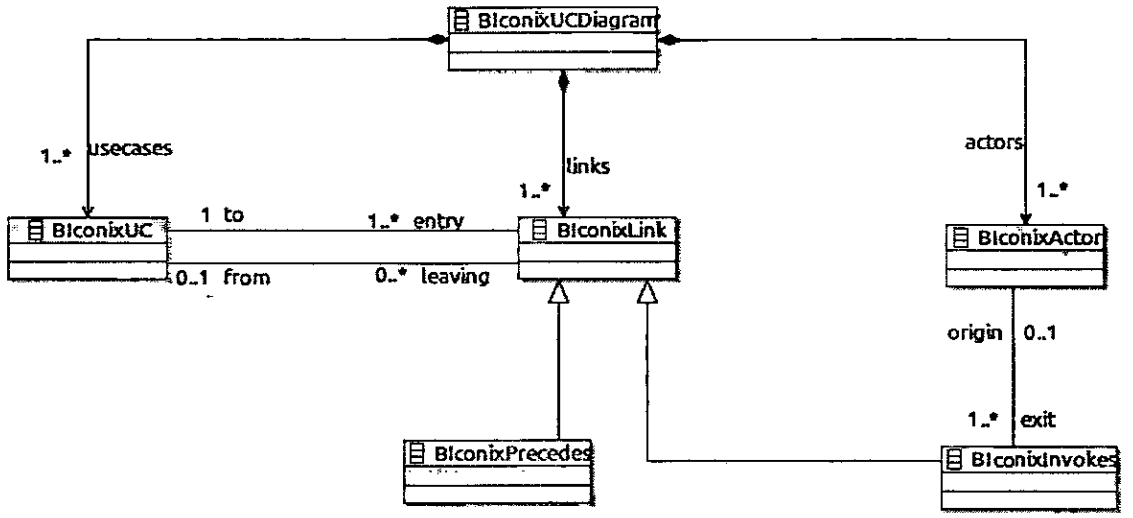


Figura 13: BIconix – Meta-modelo do Diagrama de Casos de Uso. Fonte: o autor

5.2 Descrição das Meta-classes

5.2.1 BIconixUCDiagram

A BIconixUCDiagram é uma meta-classe que representa um Diagrama de Casos de Uso, sendo ela a responsável por armazenar todos os outros elementos do mesmo, porém, não possui representação gráfica. Ela possui três associações (*usecases*, *links* e *actors*), obrigando que cada diagrama criado possua, no mínimo, um objeto instanciado para cada um de seus elementos. Além disso, ela deve obrigatoriamente possuir um nome.

- usecases:BIconixUC[1..*], representa um conjunto que guarda todas as instâncias dos casos de uso.
- links:BIconixLink[1..*], representa um conjunto que guarda todas as relações existentes entre casos de uso, ou entre casos de uso e atores.
- actors:BIconixActor[1..*], representa um conjunto que guarda todas as instâncias dos atores.

5.2.2 BIconixUC

A BIconixUC é uma meta-classe que representa um Caso de Uso. Sua representação gráfica se dá, da mesma forma que a utilizada na UML, através de uma

elipse contendo a descrição do caso de uso em seu centro. Todas as instâncias desse elemento devem possuir um nome associado, onde o mesmo deve ser único no diagrama, e seus dois conjuntos (*leaving* e *entry*) não podem conter nenhum elemento em comum.

- leaving:BIconixLink[1..*], esse conjunto armazena todos os relacionamentos que saem de um BIconixUC.
- entry:BIconixLink[1..*], semelhante ao *leaving*, porém, armazena todos os relacionamentos que chegam a um BIconixUC.

5.2.3 BIconixLink

A meta-classe BIconixLink representa os relacionamentos existentes entre dois ou mais Casos de Uso, ou entre, no mínimo, um Caso de Uso e um Ator. Esta meta-classe não possui representação gráfica, servindo apenas como uma super-classe para os únicos dois relacionamentos existentes, *precedes* ou *invokes*. Ela possui apenas uma restrição: não pode existir auto-relacionamento entre os elementos que a utilizam, ou seja, seus dois atributos (*from* e *to*) devem ser diferentes.

- from:BIconixUC[0..1], este atributo armazena o Caso de Uso que originou a ligação. Podemos notar que ele pode conter no máximo um Caso de Uso associado, pois uma ligação não pode se originar de vários Casos de Uso, além disso, se a ligação se originar de um Ator, este atributo ficará vazio.
- to:BIconixUC[1], este atributo armazena o Caso de Uso ao qual a ligação se destina, ou seja, o Caso de Uso alvo.

5.2.4 BIconixInvokes

A BIconixInvokes é uma meta-classe que representa uma ligação do tipo Invokes, proposto pelo Iconix, entre os Casos de Uso e/ou Atores. Ela engloba os dois esteriótipos (extends e includes) usados na UML, além de não permitir que vários Atores "usem" um caso de uso simultaneamente. Sua representação gráfica se dá através de uma linha tracejada contendo uma seta simples na extremidade, em que se destina a ligação, e é nomeada <<invokes>>. Esta meta-classe possui uma generalização com a BIconixLink, ou seja, herda todas as suas propriedades e restrições, e possui um outro atributo chamado de

origin.

- Origin:BIconixActor[0..1], armazena o Ator que originou a ligação. Se há uma ligação do tipo Invokes entre um Ator e um Caso de Uso, então este atributo deve armazenar aquele Ator.

5.2.5 BIconixPrecedes

A BIconixPrecedes é uma meta-classe que representa uma ligação do tipo Precedes, proposto pelo Iconix, entre os Casos de Uso, indicando que um deve ser finalizado para que o outro seja iniciado. Sua representação gráfica se dá de forma semelhante a da BIconixInvokes, com a diferença que ela é nomeada como <<precedes>>. Esta meta-classe possui uma generalização com a BIconixLink, ou seja, herda todas as suas propriedades e restrições.

5.2.6 BIconixActor

A meta-classe BIconixActor representa um Ator. No processo BIconix, assim como na UML, possibilita a identificação de um usuário, ou de um outro sistema, que interage com o software através dos Casos de Uso que a ele estão associados. Sua representação gráfica se dá igualmente a usada pelo OMG, por meio de um boneco com um nome, que o identifica unicamente, logo abaixo. Ela possui duas restrições: deve possuir um nome, e não pode ser o elemento alvo de uma ligação BIconixInvokes. Esta meta-classe possui um único conjunto, chamado *exit*:

- exit:BIconixInvokes[1..*], armazena todos as ligações do tipo Invokes que se originam dela.

5.3 Criação Do Editor Gráfico

Com a definição dos meta-modelos concluída, iremos, agora, criar o nosso editor gráfico. O primeiro passo foi criar um arquivo no qual modelamos o nosso diagrama. A Figura 15, mostra o resultado final da criação do arquivo "ecorediag" (Ecore Diagram), que faz parte do EMF, assim como o arquivo "ecore", citados na seção 4.1 do

capítulo 4. Por meio do "ecore" iremos gerar outros cinco arquivos que irão auxiliar nosso desenvolvimento. Devemos ressaltar que, todas as alterações que são realizadas nos meta-modelos do "ecore" são automaticamente aplicadas ao "ecorediag", e vice-versa.

O GMF dispões de um "dashboard", Figura 14, que nos permite gerar esses arquivos sem precisar codificá-los. Porém, algumas vezes, a edição de alguns desses arquivos se faz necessária, uma vez que a lógica aplicada pelo "dashboard" não é satisfatória.

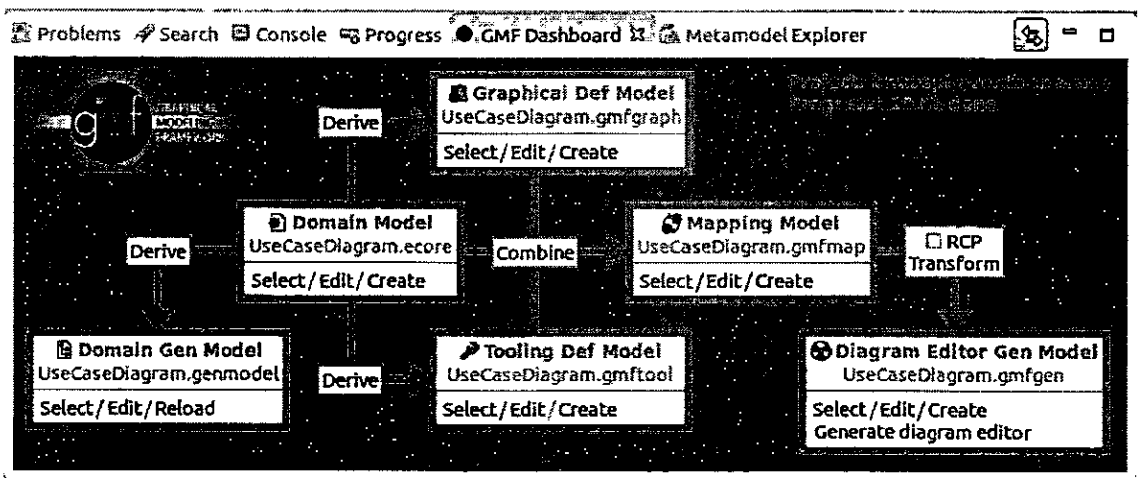


Figura 14: *GMF Dashboard.Fonte: o autor*

Nosso segundo passo é derivar o arquivo "genmodel" (Generator Model) do modelo de domínio, nosso arquivo "ecore". Ele contém todas as informações necessárias para a criação dos códigos fontes, como classes e interfaces, bem como a lógica de negócio do nosso diagrama. Ademais, ele possibilita a criação de outros dois projetos que são utilizados para a edição do diagrama, e outro para testes.

O próximo passo será criar outros dois arquivos que se fazem necessários para a geração de um terceiro. O primeiro é o "gmfgraph", que contém todas as definições gráficas dos nossos meta-modelos criados no "ecore", e onde efetuamos todas as alterações gráficas desejadas. O segundo é o "gmftool", que contém um conjunto de atalhos que possibilita a instanciação e criação dos elementos do diagrama. Após a criação desses dois arquivos, iremos combiná-los para gerar o "gmfmap", um arquivo de configuração criado para mapear os meta-modelos, seus elementos gráficos e seus controladores.

Por fim, vamos transformar todos os arquivos já criados até o momento para gerar, ainda por meio do "dashboard", o "gmfgen", que é responsável por criar os códigos do editor gráfico.

Como dito anteriormente, alguns desses arquivos precisam ser modificados para que se possa obter o resultado final esperado. Para a criação desta ferramenta, algumas modificações foram realizadas no código gerado pelo "gmfmodel", porém, o arquivo não sofreu alteração alguma. Entretanto, não abordaremos os detalhes destas alterações pois o objetivo deste trabalho não é a criação de um tutorial para uso das tecnologias citadas.

A Figura 15 mostra o resultado final do editor gráfico já finalizado, e sendo executado na plataforma Rodin, e possibilitando a criação e edição dos elementos do Diagrama de Casos de Uso. Na próxima subseção iremos mostrar as regras que irão possibilitar a transformação do diagrama para o Event-B.

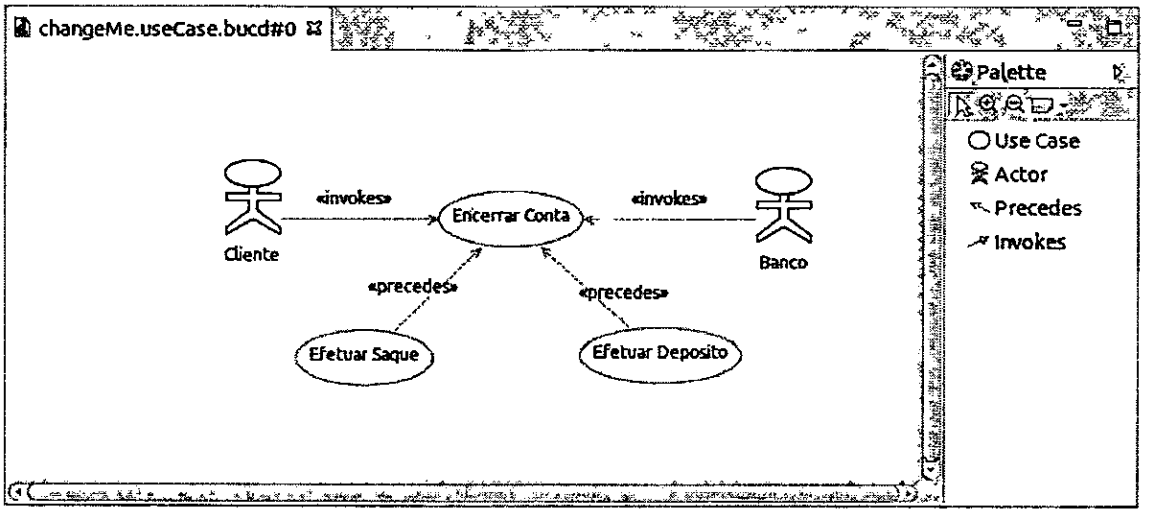


Figura 15: Editor gráfico – Diagrama de Casos de Uso. Fonte: o autor

Na seção a seguir, iremos discorrer sobre as regras propostas que serão aplicadas para a transformação do Diagrama de Casos de Uso em meta-modelos do Event-B.

5.4 Regras de Transformação

Após a definição dos meta-modelos do diagrama, podemos agora partir para a definição das regras de transformação. Os diagramas criados no nosso editor servirão de entrada para as regras. Em contra partida, instâncias dos meta-modelos do Event-B serão obtidos como saída da transformação. As regras descritas aqui, serão apresentadas de forma simples e através de exemplos. Para maiores detalhes, a definição completa delas em QVTO encontra-se no Apêndice A.

A fim de facilitar o mapeamento entre as duas linguagens, utilizamos herança entre alguns meta-modelos do Diagrama de Casos de Uso (BIconixUC, BIconixActor, BIconixLink, etc) com os meta-modelos do Event-B, tais como *Diagram* e *AbstractExtension*, e outras meta-classes que possuem EventB como prefixo (EventBNamed, EventBElement, etc). O meta-modelo completo dos elementos do Event-B pode ser encontrado em [31].

5.4.1 Diagrama de Casos de Uso em Event-B

A seguir, serão apresentadas os elementos correspondentes em Event-B para cada elemento do Diagrama de Casos de Uso.

5.4.1.1 BIconixUCDiagram em Event-B

Toda instância do BIconixUCDiagram gera uma instância de uma *Machine* no mapeamento, ambas possuindo o mesmo nome, além de referenciar um *Context* que será gerado pelo mapeamento do BIconixClassDiagram, o qual não será abordado neste trabalho, e não possui nenhuma inicialização no evento *INITIALIZATION*. No *Context* referenciado, o mapeamento ainda cria três construções: uma *CarrierSet* chamada de *Status*; três *Constants* (nostatus, started e ended); e um *Axiom* enumerando as constantes em estados possíveis do conjunto *Status*.

Na Figura 16 podemos ver um exemplo de um Diagrama de Casos de Uso chamado "EncerrarConta", e sua respectiva notação em Event-B.

MACHINE EncerrarConta
SEES
EncerrarContaPoupanca
EVENTS
INITIALIZATION
BEGIN
END
END
SETS
Status
CONSTANTS
nostatus
started
ended
AXIOMS
axm1 : partition(Status, {nostatus}, {started}, {ended})

Figura 16: BIconixUCDiagram em Event-B. Fonte: o autor

5.4.1.2 BIconixUC em Event-B

Cada instância da BIconixUC gera uma instância de *Constant* chamada "waitingfor{BIconixUC.name}", no *Context* gerado pelo BIconixUCDiagram, onde é adicionada ao conjunto Status. Além disso, várias instâncias são criadas dentro da *Machine:*

- Uma *Variable* chamada "control_{BIconixUC.name}".
- Uma *Invariant,* "control_{BIconixUC.name} ∈ Status", indicando o status de uma variável.
- Ao evento *INITIALIZATION* é adicionada uma *Action* chamada "control_{BIconixUC.name} := nostatus", indicando que não existe um status inicial, e

- Um *Event* chamado {BIconixUC.name}, que indica o fim da execução do Caso de Uso. A ele ainda adicionamos:
  - uma *Guard* chamada "control_{BIconixUC.name} = started", e
  - uma *Action* chamada "control_{BIconixUC.name} := ended"

    Este evento possui ainda algumas alterações em alguns de seus atributos, porém, eles apenas serão visíveis na codificação das regras.
  - convergence = ordinary
  - extended = false

Na Figura 17, vemos um exemplo de um Caso de Uso e seu respectivo elemento em Event-B.

CONSTANTS

waitingforEncerrarContaPoupanca

AXIOMS

axm1 : partition(Status, {nostatus}, {started}, {ended}, {waitingforEncerrarContaPoupanca})

VARIABLES

control_EncerrarContaPoupanca

INVARIANTS

inv1 : control_EncerrarContaPaupanca ∈ Status

EVENTS

INITIALIZATION

BEGIN

act1 : control_EncerrarContaPaupanca := nostatus

END

EVENT EncerrarContaPaupanca ≙

WHEN

grd1 : control_EncerrarContaPaupanca = started

THEN

act2 : control_EncerrarContaPaupanca := ended

END

Figura 17: BIconixUC em Event-B. Fonte: o autor

5.4.1.3 BIconixLink em Event-B

As instâncias da BIconixLink não são mapeadas, pois servem apenas como uma base para os esteriótipos Invokes e Precedes, por tanto, não possuem elementos equivalentes no Event-B.

5.4.1.4 BIconixInvokes em Event-B

As instâncias da BIconixInvokes, dentro de uma *Machine*, podem gerar instâncias diferentes de acordo com algumas condições. A primeira ocorre quando um caso de uso invoca um outro, neste caso temos um *Event*, chamado de "{BIconixInvokes.from.name}invokes{BIconixInvokes.to.name}", dentro dele ainda temos:

- convergence = ordinary; e extended = false
- duas *Guard*, indicando as condições de invocação do Caso de Uso:
  - "control_{BIconixInvokes.from.name} = started"
  - "control_{BIconixInvokes.to.name} = nostatus"
- e duas Actions, indicando o status do Caso de Uso após ser inicializado:
  - "control_{BIconixInvokes.from.name} := waitingfor{BIconixInvokes.to. name}", e
  - "control_{BIconixInvokes.to.name} := started"

A segunda condição ocorre quando um Caso de Uso invoca mais de um Caso de Uso. As instâncias geradas são semelhantes, diferindo apenas na *Guard* "control_{BIconixInvokes.from.name} = started", onde são inseridas disjunções para cada Caso de Uso invocado, onde as mesmas serão também inseridas na *Guard* de mesmo nome para o Caso de Uso invocador:

- "control_{BIconixInvokes.from.name} = started ∨ (control_{BIconixInvokes. from.name} = waitingfor{BIconixUC.leaving[!{BIconixInvokes.to.name}].to. name}) ∧ control_{BIconixUC.leaving[!{BIconixInvokes.to.name}].to.name} = ended" .

Na Figura 18 vemos um exemplo de um Caso de Uso, "GerenciarConta", invoca outros dois Casos de Uso (Login e EncerrarConta)

EVENTS

EVENT GerenciarContainvokesEncerrarConta ≙

WHEN

grd2 : control_GerenciarConta = started ∨ (control_GerenciarConta = waitingforLogin ∧ control_Login = ended)

THEN

END

EVENT GerenciarContainvokesLogin ≙

WHEN

grd3 : control_GerenciarConta = started ∨ (control_GerenciarConta = waitingforEncerrarConta ∧ control_EncerrarConta = ended)

THEN

END

EVENT GerenciarConta ≙

WHEN

grd4 : control_GerenciarConta = started ∨ (control_GerenciarConta = waitingforLogin ∧ control_Login = ended) ∨ (control_GerenciarConta = waitingforEncerrarConta ∧ control_EncerrarConta = ended)

THEN

END

Figura 18: *BIconixInvokes em Event-B. Fonte: o autor*

5.4.1.5 BIconixPrecedes em Event-B

As instâncias da BIconixPrecedes gera algumas instâncias dentro de uma Machine:

- uma *Variable* chamada {BIconixPrecedes.from,name}_precedes para o caso de uso que originou a ligação
- uma *Invariant* chamada "{BIconixPrecedes.from.name}_precedes ∈ Status"
- uma *Action* chamada de "{BIconixPrecedes.from.name}_precedes := nostatus",

dentro do evento *INITIALIZATION*, indicando que não existe um status inicial.

- uma *Guard* chamada "{BIconixPrecedes.from.name}_precedes = nostatus", dentro do evento que representa o Caso de Uso que originou a ligação, indicando que ainda não foi finalizada. E ainda dentro deste evento,
- uma *Action* chamada "{BIconixPrecedes.from.name}_precedes := ended", indicando que foi finalizado. E
- uma *Guard* chamada de "{BIconixPrecedes.from.name}_precedes = ended", para cada evento que invocou o Caso de Uso alvo desta ligação.

A Figura 19 a seguir, mostra um exemplo de um Caso de Uso chamado "Login" precede um outro chamado "EncerrarConta", que por sua vez, é invocado apenas pelo Caso de Uso "GerenciarConta".

VARIABLES

Login_precedes

INVARIANTS

inv2 : Login_precedes ∈ Status

EVENTS

INITIALIZATION

BEGIN

act3 : Login_precedes := nostatus

END

EVENT Login ≙

WHEN

grd5 : Login_precedes = nostatus

THEN

act4 : Login_precedes := ended

END

EVENT GerenciarContainvokesEncerrarConta ≙

WHEN

grd6 : Login_precedes = ended

THEN

END

Figura 19: BIconixPrecedes em Event-B. Fonte: o autor

5.4.1.5 BIconixActor em Event-B

As instâncias mapeadas da BIconixActor são parecidas com as da BIconixUC, apresentando algumas diferenças.

- No *Context*, diferente da BIconixUC, nada é gerado.
- uma *Action* chamada "control_{BIconixActor.name} := ended" é gerada dentro do evento *INITIALIZATION*.
- um *Event* chamado "{BIconixActor.name}" é gerado com duas *Action*:
  - A primeira chamada "control_{BIconixUC.name} := nostatus", para os Casos de Uso existentes, e
  - a segunda para cada ligação do tipo Precedes existente, chamada "{BIconixPrecedes.from.name}_precedes := nostatus"

VARIABLES
control_Cliente
INVARIANTS
inv7 : control_Cliente ∈ Status
EVENTS
INITIALIZATION
BEGIN
act5 : control_Cliente := ended
END
EVENT Clientestarts ≙
WHEN
grd7 : control_Cliente = ended
THEN
act6 : control_Cliente := started
END
EVENT Cliente ≙
WHEN
grd8 : control_Cliente = started
THEN
act7 : control_Cliente := ended
act8 : control_EncerrarConta := nostatus
END

Figura 20: BIconixActor em Event-B. Fonte: o autor

- uma segunda *Event* é criada indicando que um Ator foi iniciado, chamada de "{BIconixActor.name}starts", contendo uma *Guard*, chamada "control_{BIconixActor.name} = ended", e uma *Action* chamada "control_{BIconixUC.name} := started".

Na Figura 20, podemos ver um exemplo de um Ator, chamado de Cliente, e seu correspondente em Event-B.

6 CONSIDERAÇÕES FINAIS

O objetivo deste trabalho foi apresentar uma proposta para reduzir a ambiguidade do Diagrama de Casos de Uso do Iconix, derivado da UML. O método formal Event-B foi somado ao Iconix com o intuito de proporcionar a verificação dos requisitos iniciais com a análise de software, por meio de provas matemáticas.

Neste trabalho também mostramos que o processo Iconix proporciona a diminuição do tempo da análise de software, bem como os erros de análise, graças a sua metodologia fácil e simples, porém, bastante robusta.

Vimos também que o uso de métodos formais, ainda na fase de análise do projeto, possibilitam que o desenvolvimento de um software não sofra o retardamento causado pelos erros proveniente de uma análise mal realizada. Porém, apesar de permitirem a criação de softwares mais confiáveis, muitos gerentes de desenvolvimento ainda são avessos em utilizá-los, muitas vezes pelo custo inicial de sua aplicação tornar-se mais elevado, muito embora, o custo total do projeto seja mais baixo.

No decorrer deste trabalho foram encontradas algumas dificuldades com relação a criação da ferramenta CASE, dentre elas, destaca-se o uso do GMF. No GMF, a maior dificuldade se concentra na criação dos arquivos de configuração dos projetos por meio do *dashboard*, que, apesar de facilitar a geração dos mesmos, ainda apresenta alguns problemas referentes a manipulação dos componentes gráficos e o mapeamento com seus controladores, e a paleta do menu, por exemplo. Ademais, a edição de alguns dos códigos fontes gerados fez-se necessária para que os componentes se adequassem a especificação inicial da ferramenta. Além disso, a forte ligação existente entre os arquivos retardou o desenvolvimento, pois, algumas alterações feitas refletiam em uma nova geração dos códigos fontes, o que ocasionava na perda das modificações realizadas

6.1 Trabalhos Futuros

Este trabalho faz parte de um projeto maior do consórcio ADVANCE da comunidade européia chamado BIconix, que tenta integrar os diagramas de casos de uso, modelo de domínio, diagrama de sequência e diagrama de robustez, por isso um dos trabalhos futuros é integrar o Diagrama de Casos de Uso com o Modelo de Domínio, com

o intuito de agregar a esses dois diagramas do Iconix, uma ferramenta que possa ser usada para verificar de forma mais completa a análise de sistemas.

Uma proposta para a comunidade acadêmica, é a resolução do problema da criação dos elementos gráfico do gmfgraph, que seria resolvido com a adição de um editor que proporciona, de forma gráfica, o manuseio dos componentes do gmfgraph, o que atualmente é feito por meio de um arquivo XML.

REFERÊNCIAS

[1] BOOCH, Grady et al. UML : Guia do Usuário, O mais avançado tutorial sobre Unified Modeling Language (UML), elaborado pelos próprios criadores da linguagem. 2 ed. Rio de Janeiro. Campus, 2005.

[2] ROSENBERG, D.; STEPHENS, M. Use Case Driven Object Modeling with UML: Theory and Practice. [S.l.]: Apress, 2007. ISBN 0321278275.

[3] BECK, K.; ANDRES, C. Extreme Programming Explained : Embrace Change. 2nd. ed. [S.l.]: Addison-Wesley Professional, 2004. Paperback.

[4] KRUCHTEN, P. The Rational Unified Process: An Introduction. 3. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321197704.

[5] SCHWABER, K.; BEEDLE, M. Agile Software Development with Scrum. Upper Saddle River, NJ: Prentice Hall, 2002. ISBN 978-0-13-067634-4.

[6] ABRIAL, J.-R. The B-book: assigning programs to meanings. 1st. ed. New York, NY, USA: Cambridge University Press, 1996. ISBN 0-521-49619-5.

[7] ABRIAL, J.-R. Modeling in Event-B: System and Software Engineering. 1st. ed. New York, NY, USA: Cambridge University Press, 2010. ISBN 0521895561, 9780521895569.

[8] BJØRNER, D.; JONES, C. B. The Vienna Development Method: The Meta-Language. Lecture Notes in Computer Science, Springer-Verlag, London, UK, v. 61, 1978.

[9] SPIVEY, J. The z notation - a reference manual. Prentice Hall International Series in Computer Science, Prentice Hall, p. I–XI, 1–155, 1989

[10] ALI, Mouez et al. Formal Consistency Verification of UML Requirement and Analysis Models. Formal Languages for Computer Simulation: Transdisciplinary Models and Applications. IGI Global, 2013. 205-235.

[11] HA, Il-Kyu; KANG, Byung-Wook. Meta-validation of UML structural diagrams and behavioral diagrams with consistency rules. In: Communications, Computers and signal Processing, 2003. PACRIM. 2003 IEEE Pacific Rim Conference on. IEEE, 2003. p. 679-683.

[12] IBRAHIM, Noraini et al. Use case driven based rules in ensuring consistency of UML model. AWERProcedia Information Technology and Computer Science, v. 1, 2013.

[13] CHANDA, J. et al., Traceability of Requirements and Consistency Verification of UML UseCase, Activity and Class diagram: A Formal Approach, in: International Conference on Methods and Models in Computer Science 2009 (ICM2CS), 2009, pp. 1-4

[14] SCHMID, Helmut. Efficient parsing of highly ambiguous context-free grammars with

bit vectors. In: Proceedings of the 20th international conference on Computational Linguistics. Association for Computational Linguistics, 2004. p. 162.

[15] Object Modeling Technique (OMT). Disponível em <http://www.idi.ntnu.no/grupper/su/publ/html/totland/ch0527.htm>. Acessado em: 29 mai. 2013.

[16] JACOBSON, Ivar; LINDSTRÖM, Fredrik. Reengineering of old systems to an object-oriented architecture. ACM, 1991.

[17] ROSENBERG, Doug; COLLINS-COPE, Mark; STEPHENS, Matt. Agile development with ICONIX process: people, process, and pragmatism. Apress, 2005.

[18] LUCKHAM, David C. et al. ANNA A Language for Annotating Ada Programs: Reference Manual. ISBN 3540179801

[19] Rodin User's Handbook v.2.7. Disponível em <http://handbook.event-b.org/current/html/tut_eventb_concepts.html>. Acessado em: 12 jun 2013.

[20] ABRIAL, Jean-Raymond. HALLERSTEDE, Stefan. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. ETH Zurich, Switzerland. IOS Press. 2007

[21] The B-Toolkit. Disponível em: <http://web.archive.org/web/20041012141220/http://www.b-core.com/ONLINEDOC/BToolkit.html>. Acessado em 16 Jun 2013.

[22] ABRIAL, J.-R.; BUTLER, M.; HALLERSTEDE, S.; VOISIN, L. An open extensible tool environment for Event-B. In: Proceedings of the International Conference on Formal Engineering Methods (ICFEM 2006). [S.l.]: Springer, 2006. (Lecture Notes in Computer Science), p. 588–605.

[23] Deploy. Disponível em <www.deploy-project.eu>. Acessado em: 20 jun 2013.

[24] (STEINBERG et al., 2009)
STEINBERG, Dave. et. al. EMF: Eclipse Modeling Framework. 2 ed. ISBN 021331885. 2009.

[25] GEF (Graphical Editing Framework). Disponível em <http://www.eclipse.org/gef/>. Acessado em: 23 mai. 2013.

[26] Draw2d. Disponível em <http://www.eclipse.org/gef/draw2d/>. Acessado em: 23 mai. 2013.

[27] Graphical Modeling Project (GMP). Disponível em <http://wiki.eclipse.org/GEF>. Acessado em: 25 mai. 2013.

[28] RÖHS, Malte. A visual editor for semantics specifications using the eclipse graphical modeling framework. Dissertação de Mestrado, University of Paderborn. 2008

[29] DI RUSCIO, Davide; LÄMMEL, Ralf; PIERANTONIO, Alfonso. Automated co-evolution of GMF editor models. In: Software Language Engineering. Springer Berlin Heidelberg, 2011. p. 143-162.

[30] GARDNER, Tracy et al. A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard. In: MetaModelling for MDA Workshop. 2003. p. 178-197.

[31] SNOOK, C., FRITZ, F. and ILLISAOV, A. (2010) An EMF Framework for Event-B. At Workshop on Tool Building in Formal Methods - ABZ Conference, Orford, Quebec, Canada.

[32] DE SOUSA, T. C. . Introdução ao Iconix: Um processo de desenvolvimento prático baseado no RUP. SQL magazine, , v. 94, p. 58 - 66, 01 dez. 2011.

APÊNDICE A

Neste apêndice encontram-se as regras de transformação que foram implementadas em QVTO e descritas no capítulo 5.

```
//*************************   CARRIER SETS SECTION
*****************************************

mapping usecasediagram::BIconixUCDiagram::usecase2sets(in generatorID :
String) : Sequence(eventb::context::CarrierSet) {
    init {
        result := self.usecases[usecasediagram::BIconixUC].map
usecase2set(generatorID)->asSequence()->
            union(self.actors[usecasediagram::BIconixActor].map
actor2set(generatorID)->asSequence())->
            union(self.map status2carrierSet()->asSequence());
    }
}

mapping usecasediagram::BIconixUCDiagram::status2carrierSet() :
eventb::context::CarrierSet {
      name := STATUS;
}

mapping usecasediagram::BIconixUC::usecase2set(in generatorID : String)
:eventb::context::CarrierSet
when { self._refines = null }
{
    name := self.name + _SET;
    generated := true;
    attributes += getGeneratedAttr(generatorID);
}

mapping usecasediagram::BIconixActor::actor2set(in generatorID : String)
:eventb::context::CarrierSet
when { self._refines = null }
{
    name := self.name + _SET;
    generated := true;
    attributes += getGeneratedAttr(generatorID);
}

//*************************   AXIOMS SECTION
*************************************

mapping usecasediagram::BIconixUCDiagram::usecasediagram2axioms(in
generatorID : String) : Sequence(eventb::context::Axiom) {
    init {
        result := self.map partition2axiom()->asSequence();
    }
}
mapping usecasediagram::BIconixUCDiagram::partition2axiom() :
eventb::context::Axiom {
      name := getAxiomName();
      predicate := self.useCase2axiom() + B_RPAR;
```

```
}
helper usecasediagram::BIconixUCDiagram::useCase2axiom() : String {
      self.usecases[usecasediagram::BIconixUC]->forEach(i){
      SET_AXIOMS := SET_AXIOMS + addNewAxiom( WAITINGFOR + i.name );
      };
      return SET_AXIOMS;
}
helper addNewAxiom(in value : String) : String {
      if (value <> null) then{
         return B_COM + B_LBRC + value + B_RBRC;
    }endif;
    return "";
}


//**************************   VARIABLES SECTION
****************************************

mapping usecasediagram::BIconixUCDiagram::useCaseDiagram2variables(in
generatorID : String) : Sequence(eventb::machine::Variable) {
    init {
        result := self.usecases[usecasediagram::BIconixUC].map
useCase2variables(generatorID)->
            union(self.actors[usecasediagram::BIconixActor].map
actor2variables(generatorID))->
            union(self.links[usecasediagram::BIconixPrecedes].map
precedes2variables(generatorID));
    }
}
mapping usecasediagram::BIconixUC::useCase2variables(in generatorID :
String) : Sequence(eventb::machine::Variable) {
    init {
        result := self.map useCase2variable(generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixUC::useCase2variable(in generatorID :
String) : eventb::machine::Variable {
    name := CONTROL + self.name;
    generated := true;
    attributes += getGeneratedAttr(generatorID);
}

mapping usecasediagram::BIconixActor::actor2variables(in generatorID :
String) : Sequence(eventb::machine::Variable) {
    init {
        result := self.map actor2variable(generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixActor::actor2variable(in generatorID :
String) : eventb::machine::Variable {
    name := CONTROL + self.name;
    generated := true;
    attributes += getGeneratedAttr(generatorID);
}

mapping usecasediagram::BIconixPrecedes::precedes2variables(in
generatorID : String) : Sequence(eventb::machine::Variable) {
```

```
    init {
        result := self.map precedes2variable(generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixPrecedes::precedes2variable(in generatorID
: String) : eventb::machine::Variable {
    name := self._from.name + _PRECEDES;
    generated := true;
    attributes += getGeneratedAttr(generatorID);
}

//*************************   INVARIANTS SECTION
**************************************

mapping usecasediagram::BIconixUCDiagram::useCaseDiagram2invariants(in
generatorID : String) : Sequence(eventb::machine::Invariant) {
    init {
        result := self.usecases[usecasediagram::BIconixUC].map
useCase2invariant(generatorID)->sortedBy(i | i.name)->asSequence()->
            union(self.actors[usecasediagram::BIconixActor].map
actor2invariant(generatorID)->sortedBy(i | i.name)->asSequence())->
            union(self.links[usecasediagram::BIconixPrecedes].map
precedes2invariant(generatorID)->sortedBy(i | i.name)->asSequence());
    }
}

mapping usecasediagram::BIconixUC::useCase2invariant(in generatorID :
String) : Sequence(eventb::machine::Invariant){
    init {
        result := getInvariant(getInvariantName(), CONTROL + self.name +
B_IN + STATUS, generatorID)->asSequence()
    }
}
mapping usecasediagram::BIconixActor::actor2invariant(in generatorID :
String) : Sequence(eventb::machine::Invariant){
    init {
        result := getInvariant(getInvariantName(), CONTROL + self.name +
B_IN + STATUS, generatorID)->asSequence()
    }
}
mapping usecasediagram::BIconixPrecedes::precedes2invariant(in
generatorID : String) : Sequence(eventb::machine::Invariant){
    init {
        result := getInvariant(getInvariantName(), self._from.name +
_PRECEDES + B_IN + STATUS, generatorID)->asSequence()
    }
}

//*************************   CONSTANTS SECTION
**************************************

mapping usecasediagram::BIconixUCDiagram::usecasedagram2constants(in
generatorID : String) : Sequence(eventb::context::Constant) {
    init {
        result := self.map nostatus2constant(generatorID)->asSequence()->
            union(self.map
started2constant(generatorID)->asSequence())->
```

```
            union(self.map ended2constant(generatorID)->asSequence())->
            union(self.usecases[usecasediagram::BIconixUC].map
useCases2constant(generatorID)->asSequence());
    }
}

mapping usecasediagram::BIconixUCDiagram::nostatus2constant(in
generatorID : String) : eventb::context::Constant{
        name := NOSTATUS;
        comment := "Indica a inexistência de status";
        generated := true;
}
mapping usecasediagram::BIconixUCDiagram::started2constant(in generatorID
: String) : eventb::context::Constant{
        name := STARTED;
        comment := "Indica que o elemento foi inicializado";
        generated := true;
}
mapping usecasediagram::BIconixUCDiagram::ended2constant(in generatorID :
String) : eventb::context::Constant{
        name := ENDED;
        comment := "Indica que o elemento foi finalizado";
        generated := true;
}
mapping usecasediagram::BIconixUC::useCases2constant(in generatorID :
String) : eventb::context::Constant{
        name := WAITINGFOR + self.name;
        generated := true;
}

mapping usecasediagram::BIconixUC::useCase2constant(in generatorID :
String) : eventb::context::Constant
when {self.name = null}{
    init {
        result := getConstant( WAITINGFOR + self.name )
    }
}

//**************************   EVENTS SECTION
****************************************

mapping usecasediagram::BIconixUCDiagram::useCaseDiagram2events(in
generatorID : String) : Sequence(eventb::machine::Event) {
    init {
        result := self.map
useCase2initialization(generatorID)->asSequence()->
            union(self.actors[usecasediagram::BIconixActor].map
actor2initEvents(self, generatorID))->
            union(self.actors[usecasediagram::BIconixActor].map
actor2initEventsTwo(self, generatorID))->
            union(self.usecases[usecasediagram::BIconixUC].map
useCase2initEvents(self, generatorID))->
            union(self.links[usecasediagram::BIconixInvokes].map
invokes2initEvents(self, generatorID))->
            union(self.links[usecasediagram::BIconixInvokes].map
invokesActors2initEvents(self, generatorID));
    }
```

```
}
mapping usecasediagram::BIconixUCDiagram::useCase2initialization(in
generatorID : String) : eventb::machine::Event
{
    name := INITIALISATION;
}

mapping usecasediagram::BIconixUC::useCase2initEvents(in diagram :
usecasediagram::BIconixUCDiagram, in generatorID : String) :
Sequence(eventb::machine::Event) {
    init {
        result := self.map useCase2initEvent(diagram,
generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixUC::useCase2initEvent(in diagram :
usecasediagram::BIconixUCDiagram, in generatorID : String) :
eventb::machine::Event
{
    name := self.name;
    guards := self.map useCase2guard(diagram,
generatorID)->asSequence()->
        union(diagram.links[usecasediagram::BIconixPrecedes].map
precedes2guards(self, generatorID)->asSequence());
    actions := self.map useCase2action(generatorID)->asSequence();
    generated := true;
    convergence := core::machine::Convergence::ordinary;
    extended := false;
}
mapping usecasediagram::BIconixUC::useCase2guard(in diagram :
usecasediagram::BIconixUCDiagram, in generatorID : String) :
eventb::machine::Guard{
    name := getGuardName();
    predicate :=  CONTROL + self.name + B_EQ + STARTED +
diagram.links2disjuncts(self, self);
}
mapping usecasediagram::BIconixPrecedes::precedes2guards(in usecase :
usecasediagram::BIconixUC, in generatorID : String) :
eventb::machine::Guard
      when {self._from = usecase}
{
    init{
        result := self.map precedes2guard(generatorID);
    }
}
mapping usecasediagram::BIconixPrecedes::precedes2guard(in generatorID :
String) : eventb::machine::Guard{
    name := getGuardName();
    predicate :=  self._from.name + _PRECEDES + B_EQ + ENDED;
}

mapping usecasediagram::BIconixUC::useCase2action(in generatorID :
String) : eventb::machine::Action {
    name := getActName() ;
    action := CONTROL + self.name + B_BEQ + ENDED;
}
```

```
mapping usecasediagram::BIconixActor::actor2initEvents(in diagram :
usecasediagram::BIconixUCDiagram, in generatorID : String) :
Sequence(eventb::machine::Event) {
    init {
        result := self.map actor2initEvent(diagram,
generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixActor::actor2initEvent(in diagram :
usecasediagram::BIconixUCDiagram, in generatorID : String) :
eventb::machine::Event
{
    name := self.name;
    actions := diagram.usecases[usecasediagram::BIconixUC].map
actorEvent2action(generatorID)->asSequence()->
         union(diagram.links[usecasediagram::BIconixPrecedes].map
actorPrecedesEvent2action(generatorID)->asSequence());
    generated := true;
    convergence := core::machine::Convergence::ordinary;
    extended := false;
}
mapping usecasediagram::BIconixUC::actorEvent2action(in generatorID :
String) : Sequence(eventb::machine::Action) {
    init{
      result := self.map actorUC2events(generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixUC::actorUC2events(in generatorID :
String) : eventb::machine::Action {
    name := getActName();
    action := CONTROL + self.name + B_BEQ + NOSTATUS;
}
mapping usecasediagram::BIconixPrecedes::actorPrecedesEvent2action(in
generatorID : String) : Sequence(eventb::machine::Action) {
    init{
      result := self.map actorPrecedes2events(generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixPrecedes::actorPrecedes2events(in
generatorID : String) : eventb::machine::Action {
    name := getActName();
    action := self._from.name + _PRECEDES + B_BEQ + NOSTATUS;
}

mapping usecasediagram::BIconixActor::actor2initEventsTwo(in diagram :
usecasediagram::BIconixUCDiagram, in generatorID : String) :
Sequence(eventb::machine::Event) {
    init {
        result := self.map actor2initEventTwo(diagram,
generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixActor::actor2initEventTwo(in diagram :
usecasediagram::BIconixUCDiagram, in generatorID : String) :
eventb::machine::Event
{
    name := self.name + STARTS;
```

```
    guards := self.map actor2guard(generatorID)->asSequence();
    actions := diagram.usecases[usecasediagram::BIconixUC].map
actorEvent2actionTwo(generatorID)->asSequence();
    generated := true;
    convergence := core::machine::Convergence::ordinary;
    extended := false;
}
mapping usecasediagram::BIconixActor::actor2guard(in generatorID :
String) : eventb::machine::Guard{
    name := getGuardName();
    predicate :=  CONTROL + self.name + B_EQ + ENDED;
}
mapping usecasediagram::BIconixUC::actorEvent2actionTwo(in generatorID :
String) : Sequence(eventb::machine::Action) {
    init{
      result := self.map actorUC2eventsTwo(generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixUC::actorUC2eventsTwo(in generatorID :
String) : eventb::machine::Action {
    name := getActName();
    action := CONTROL + self.name + B_BEQ + STARTED;
}

mapping usecasediagram::BIconixInvokes::invokes2initEvents(in diagram :
usecasediagram::BIconixUCDiagram, in generatorID : String) :
Sequence(eventb::machine::Event)
when{self._from <> null}{
    init {
        result := self.map invokes2initEvent(diagram,
generatorID)->asSequence();
    }
}

mapping usecasediagram::BIconixInvokes::invokes2initEvent(in diagram :
usecasediagram::BIconixUCDiagram, in generatorID : String) :
eventb::machine::Event
{
    name := self._from.name + INVOKES + self.to.name;
    guards := self.map invokes2guardOne(diagram,
generatorID)->asSequence()->
        union(self.map invokes2guardTwo(generatorID)->asSequence())->
        union(diagram.links[usecasediagram::BIconixPrecedes].map
invokesPrecedes2guards(self.to, generatorID)->asSequence());

    actions := self.map invokes2action(generatorID)->asSequence()->
         union(self.map invokes2actionTwo(generatorID)->asSequence());
    generated := true;
    convergence := core::machine::Convergence::ordinary;
    extended := false;
}
mapping usecasediagram::BIconixPrecedes::invokesPrecedes2guards(in
usecase : usecasediagram::BIconixUC, in generatorID : String) :
eventb::machine::Guard
    when {self.to = usecase}
{
    init{
```

```
        result := self.map invokesPrecede2guard(generatorID);
    }
}
mapping usecasediagram::BIconixPrecedes::invokesPrecede2guard(in
generatorID : String) : eventb::machine::Guard{
    name := getGuardName();
    predicate :=  self._from.name + _PRECEDES + B_EQ + ENDED;
}

mapping usecasediagram::BIconixInvokes::invokes2guardOne(in diagram :
usecasediagram::BIconixUCDiagram, in generatorID : String) :
eventb::machine::Guard{
    name := getGuardName();
    predicate :=  CONTROL + self._from.name + B_EQ + STARTED +
diagram.links2disjuncts(self._from, self.to);
}
mapping usecasediagram::BIconixInvokes::invokes2guardTwo(in generatorID :
String) : eventb::machine::Guard{
    name := getGuardName();
    predicate :=  CONTROL + self.to.name + B_EQ + NOSTATUS ;
}

helper usecasediagram::BIconixUCDiagram::links2disjuncts(in _from :
usecasediagram::BIconixObject, in to : usecasediagram::BIconixUC) :
String {
    var disjunct : String := "";
    self.links[usecasediagram::BIconixInvokes]->forEach(i){
        if((_from = i._from and to <> i.to) or (_from = i._from and to =
null)) then{
            disjunct := disjunct + B_OR + B_LPAR + CONTROL + _from.name +
B_EQ + WAITINGFOR + i.to.name + B_AND + CONTROL + i.to.name + B_EQ +
ENDED + B_RPAR;
        }endif;
    };
    return disjunct;
}
helper addNewDisjunct(in value : String) : String {
    if (value <> null) then{
       return B_COM + B_LBRC + value + B_RBRC;
    }endif;
    return "";
}

mapping usecasediagram::BIconixInvokes::invokes2action(in generatorID :
String) : eventb::machine::Action {
    name := getActName();
    action := CONTROL + self._from.name + B_BEQ + WAITINGFOR +
self.to.name;
}
mapping usecasediagram::BIconixInvokes::invokes2actionTwo(in generatorID
: String) : eventb::machine::Action {
    name := getActName();
    action := CONTROL + self.to.name + B_BEQ + STARTED;
}

mapping usecasediagram::BIconixInvokes::invokesActors2initEvents(in
diagram : usecasediagram::BIconixUCDiagram, in generatorID : String) :
```

```
Sequence(eventb::machine::Event)
when{ self.origin <> null and self._from = null } {
    init {
        result := self.map invokesActor2initEvent(diagram,
generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixInvokes::invokesActor2initEvent(in diagram
: usecasediagram::BIconixUCDiagram, in generatorID : String) :
eventb::machine::Event
{
    name := self.origin.name + INVDKES + self.to.name;
    guards := self.map invokesActor2guardOne(diagram,
generatorID)->asSequence()->
        union(self.map invokesActor2guardTwo(generatorID)->asSequence());

    actions := self.map invokesActor2action(generatorID)->asSequence()->
         union(self.map
invokesActor2actionTwo(generatorID)->asSequence());
    generated := true;
    convergence := core::machine::Convergence::ordinary;
    extended := false;
}
mapping usecasediagram::BIconixInvokes::invokesActor2guardOne(in diagram
: usecasediagram::BIconixUCDiagram, in generatorID : String) :
eventb::machine::Guard{
    name := getGuardName();
    predicate :=  CONTROL + self.origin.name + B_EQ + STARTED +
diagram.linksActor2disjuncts(self.origin, self.to);
}
mapping usecasediagram::BIconixInvokes::invokesActor2guardTwo(in
generatorID : String) : eventb::machine::Guard{
    name := getGuardName();
    predicate :=  CONTROL + self.to.name + B_EQ + NOSTATUS ;
}

helper usecasediagram::BIconixUCDiagram::linksActor2disjuncts(in origin :
usecasediagram::BIconixActor, in to : usecasediagram::BIconixUC) : String
{
    var disjunct : String := "";
    self.links[usecasediagram::BIconixInvokes]->forEach(i){
        if((origin = i.origin and to <> i.to) or (origin = i.origin and
to = null)) then{
            disjunct := disjunct + B_OR + B_LPAR + CONTROL + origin.name
+ B_EQ + WAITINGFOR + i.to.name + B_AND + CONTROL + i.to.name + B_EQ +
ENDED + B_RPAR;
        }endif;
    };
    return disjunct;
}
helper addNewDisjunctActor(in value : String) : String {
    if (value <> null) then{
       return B_COM + B_LBRC + value + B_RBRC;
    }endif;
    return "";
}
```

```
mapping usecasediagram::BIconixInvokes::invokesActor2action(in
generatorID : String) : eventb::machine::Action {
    name := getActName();
    action := CONTROL + self.origin.name + B_BEQ + WAITINGFOR +
self.to.name;
}
mapping usecasediagram::BIconixInvokes::invokesActor2actionTwo(in
generatorID : String) : eventb::machine::Action {
    name := getActName();
    action := CONTROL + self.to.name + B_BEQ + STARTED;
}

//**************************   EVENTS INITIALIZATIONS SECTION
**************************************

mapping inout eventb::machine::Event::useCaseDiagram2initializations(in
useCaseDiagram : usecasediagram::BIconixUCDiagram, in generatorID :
String) {
      name := INITIALISATION;
      actions := useCaseDiagram.usecases[usecasediagram::BIconixUC].map
useCase2initializations(generatorID)->union(actions->asSequence())->
          union(useCaseDiagram.actors[usecasediagram::BIconixActor].map
actor2initializations(generatorID)->union(actions->asSequence()))->

union(useCaseDiagram.links[usecasediagram::BIconixPrecedes].map
precedes2initializations(generatorID)->union(actions->asSequence()));
}
mapping usecasediagram::BIconixUC::useCase2initializations(in generatorID
: String) : Sequence(eventb::machine::Action) {
    init {
       result := self.map
useCase2initialization(generatorID)->asSequence();
    }
}
mapping usecasediagram::BIconixUC::useCase2initialization(in generatorID
: String) : eventb::machine::Action {
      name := getActName();
    action := CONTROL + self.name + B_BEQ + NOSTATUS;
}

mapping usecasediagram::BIconixActor::actor2initializations(in
generatorID : String) : Sequence(eventb::machine::Action) {
    init {
       result := self.map
actor2initialization(generatorID)->asSequence();
    }
}

mapping usecasediagram::BIconixActor::actor2initialization(in generatorID
: String) : eventb::machine::Action {
      name := getActName();
    action := CONTROL + self.name + B_BEQ + ENDED;
}

mapping usecasediagram::BIconixPrecedes::precedes2initializations(in
generatorID : String) : Sequence(eventb::machine::Action) {
    init {
```