Instituto Superior de Engenharia do Porto

BAMoL

Formalização da linguagem e validação sintática de modelos

Nuno Filipe Sampaio Ferreira

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Arquiteturas, Sistemas e Redes

Orientador: Doutor Paulo Alexandre Gandra de Sousa

Júri:

Presidente:

Doutor José António Reis Tavares

Vogais:

Doutor Alberto António Chalupa Sampaio

Doutor Paulo Alexandre Gandra de Sousa

Porto, Outubro 2014

Aos meus avós, Joaquim e Maria Amélia

Resumo

A BAMoL (Business Application Modeling Language) é uma linguagem de domínio específico utilizada para o desenvolvimento de soluções para a plataforma myMIS, no âmbito dos sistemas de informação para a gestão. Esta linguagem carecia de dois aspetos, nomeadamente a sua formalização e a existência de mecanismos de validação sintática das soluções desenvolvidas.

Estes problemas identificados tornam impossível a validação sintática das soluções desenvolvidas utilizando esta linguagem, aumentando assim a probabilidade de existência de erros, podendo fazer com que as mesmas sejam mais ineficientes e podendo até trazer um aumento de custos de manutenção da plataforma.

De forma a resolver os problemas enunciados, foi realizada, para o primeiro, uma descrição textual de todos os constituintes da linguagem e criada uma gramática representativa da mesma, em que constam todos os seus elementos e regras. No caso do segundo problema, a sua resolução passou pela criação de uma ferramenta que utiliza a gramática criada e que permite validar sintaticamente e encontrar as falhas das soluções desenvolvidas.

Desta forma, passa a ser possível detetar os erros existentes nas soluções, permitindo assim à equipa de desenvolvimento ter maior controlo sobre as mesmas, podendo torná-las mais corretas, na perspetiva das regras da linguagem.

Palavras-chave: Linguagem de domínio específico, Modelo, Validação sintática

Abstract

BAMoL (Business Application Modeling Language) is a domain-specific language used to develop solutions for myMIS platform within management information systems. To this language are missing two things, which are its formalization and the existence of mechanisms for syntactic validation of the developed solutions.

These problems make it impossible validate the syntax of the developed solutions with this language, thus increasing the likelihood of errors in solutions. Furthermore may cause them to be more inefficient and may even increase the maintenance costs of the platform.

In order to solve the problems mentioned, was done for the first, a textual description of all the constituents of language and the creation of a grammar with all language's elements and rules. For the second problem, its resolution was done by the creation of a tool that uses the created grammar and allows syntactically validate and find the faults of the developed solutions.

Thus, it becomes possible to detect errors in existing solutions, thereby allowing the development team to have greater control over them and can make them more accurate, from the perspective of the language's rules.

Keywords: Domain-specific language, Model, Syntactic validation

Agradecimentos

Durante a realização desta tese e de todo o mestrado em que a mesma se inclui, tive a sorte de reunir o apoio de várias pessoas, que se tornaram essenciais ao longo de todo este percurso. A todos eles deixo o meu agradecimento:

À minha namorada, pela força e apoio que sempre me deu, principalmente nos momentos menos bons em que esteve presente para me ajudar a ultrapassá-los. Por nunca me ter deixado desistir e por me fazer acreditar que consigo sempre concretizar os meus objetivos.

À minha família, que me fez ser o que sou hoje e me ajudou ao longo de toda a vida a superar as muitas dificuldades que foram surgindo. Por me terem educado, cuidado sempre de mim e estado sempre ao meu lado.

Aos meus colegas de trabalho, por me ajudarem a ser um melhor profissional todos os dias e pela ajuda que me deram na realização desta tese.

Ao meu orientador, pelos conselhos e ajuda que me deu ao longo do tempo.

Ao Instituto Superior de Engenharia do Porto e ao seu Departamento de Engenharia Informática, que através do seu corpo docente tiveram um papel fulcral na minha formação académica, ajudando-me a ser o profissional que sou atualmente.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

AOM	*Adaptive Object-Model*
BAMoL	*Business Application Modeling Language*
REA	*Resources, Events and Agents*
SAFT	*Standard Audit File for Tax purposes*

1 Introdução

1.1 Problema

Neste documento serão analisadas e apresentadas soluções para duas necessidades identificadas na plataforma myMIS.

O primeiro problema identificado está relacionado com a linguagem de domínio específico utilizada na plataforma, a BAMoL (Business Application Modeling Language), e prende-se com a inexistente formalização e documentação da mesma. A falta de formalização da linguagem torna impossível a validação sintática dos modelos desenvolvidos na plataforma myMIS, enquanto a ausência de documentação torna muito difícil explicar a linguagem a pessoas externas à equipa de desenvolvimento da plataforma.

O outro problema abordado neste documento é a falta de mecanismos de validação de modelos desenvolvidos na plataforma myMIS. A inexistência destes mecanismos torna mais difícil encontrar erros de desenvolvimento que possam ter sido realizados. O ponto essencial neste caso é que os erros existentes podem trazer problemas às soluções desenvolvidas, no sentido de diminuírem a qualidade das mesmas e de aumentarem os custos de manutenção da plataforma.

1.2 Contextualização

A plataforma myMIS[1] é uma solução para desenvolvimento ágil de sistemas de informação para a gestão e que tem como base uma linguagem de domínio específico, a Business Application Modeling Language (BAMoL). Esta linguagem baseia-se na teoria económica e na teoria da contabilidade e incorpora em si os conceitos e regras necessários à criação de aplicações, dentro do domínio dos sistemas de informação de gestão.

[1] www.mymis.biz

Sendo uma linguagem de domínio específico, caracteriza-se por ser focada em resolver problemas dentro do domínio em que se insere. Este ponto distingue esta linguagem das linguagens de propósito genérico, como são o C# ou o Java.

A utilização de uma linguagem de domínio específico permite obter algumas vantagens, como a simplicidade de manutenção e a possibilidade das aplicações serem desenvolvidas por especialistas no domínio do problema, em vez de programadores [Deursen, A. *et al*., 2000; Hudak, P., 1997]. Por outro lado, o desenvolvimento e manutenção da linguagem e a dificuldade de explicar os conceitos aos desenvolvedores tornam-se desvantagens desta abordagem [Deursen, A. *et al*., 2000].

O desenvolvimento de aplicações BAMoL é baseado no desenvolvimento orientado a modelos, em que as aplicações são desenvolvidas através de modelos, em vez de código. Cada modelo permite descrever soluções com termos mais aproximados do domínio do problema, afastando assim os conceitos tecnológicos da implementação [Selic, B., 2003].

Um modelo é um conjunto de elementos que se relacionam e interagem e que respeita uma dada formalidade, definida por um meta-modelo. Esse meta-modelo define o conjunto de conceitos e regras que devem ser respeitadas no desenvolvimento dos modelos e é possível representá-lo através de uma linguagem de domínio específico.

O meta-modelo representado pela BAMoL utiliza a arquitetura adaptive object-model como base. Desta forma, a lógica de negócio é guardada na base de dados, em vez de ser implementada nas respetivas classes (numa abordagem orientada a objetos) [Yoder, J. *et al*. (2)]. Assim, as entidades, os seus respetivos comportamentos e relacionamentos são descritos (e não codificados), originando assim os meta-dados de cada aplicação, que são interpretados em runtime [Yoder, J. *et al*. (1)].

Com esta abordagem é possível criar sistemas mais flexíveis, no sentido em que alterações comportamentais podem não obrigar a alterações de código, mas apenas a uma mudança dos meta-dados.

Na BAMoL, os conceitos do meta-modelo são baseados numa framework de contabilidade denominada por “Resources, Agents and Events” (REA). Três dos seus conceitos base, aqueles que dão origem ao nome desta framework, podem definir-se da seguinte forma: Resource representa um objeto com valor económico, controlado por uma empresa e que pode ser transacionado; Agent descreve qualquer entidade que pode transacionar um Resource; Event representa a ação da troca de um Resource entre dois Agents. A estes conceitos acrescem ainda outros dois: Commitment (que representa a obrigação de um Agent realizar em Event num momento futuro) e Contract (que descreve um conjunto de Commitments e condições, que quando se verificam podem criar novos Commitments) [McCarthy, W., 1982; Hruby, P. *et al*.].

É em todas estas noções apresentadas anteriormente que assenta o objeto de estudo desta tese: a linguagem de domínio específico BAMoL.

1.3 Objetivos

Esta tese tem como objetivo propor uma solução para cada um dos problemas identificados anteriormente.

Para o problema da falta de formalização e documentação da linguagem BAMoL, pretende-se efetuar uma descrição textual da linguagem e criar uma gramática que permita materializar a formalização da BAMoL.

No caso do problema da inexistência de mecanismos de validação de modelos, é objetivo desta tese criar uma ferramenta que o permita fazer. Esta ferramenta deve ter em conta a gramática da linguagem, de forma a permitir encontrar todos os erros sintáticos de cada modelo.

1.4 Contribuições

Para o primeiro problema foi desenvolvida uma descrição de todos os elementos da BAMoL em linguagem natural, de forma a deixar bem explícito o que são e para que servem cada um desses constituintes. Para além disso, foi criada uma gramática que permite validar as aplicações desenvolvidas.

Em relação ao segundo problema, foi criada uma ferramenta que permite validar as aplicações existentes no sistema, utilizando a gramática referida anteriormente. Por não ser possível efetuar todas as validações recorrendo apenas à mesma, foi necessário inserir mecanismos adicionais na ferramenta desenvolvida, permitindo assim encontrar os erros nas aplicações existentes.

Para além das soluções apresentadas para os problemas identificados, foi também realizado um levantamento de validações inexistentes na ferramenta de desenvolvimento de modelos que, por não serem realizadas atualmente, podem levar a que existam vários erros nas soluções desenvolvidas.

Os trabalhos realizados no âmbito desta tese trazem novos meios à equipa de desenvolvimento da plataforma myMIS em dois campos diferentes. Em primeiro lugar, com a descrição textual da linguagem BAMoL e a sua formalização (através da criação da gramática representativa dos elementos da linguagem e dos seus relacionamentos) permite facilitar a explicação da linguagem a desenvolvedores externos e que desconhecem por completo a linguagem. Por outro lado, com a ferramenta de validação desenvolvida (que utiliza a gramática criada) passa a ser possível verificar se as aplicações existentes contêm erros, algo impossível de fazer sem o auxílio desta nova ferramenta. Este fator adquire uma grande importância, pois para além de ser possível encontrar pontos a retificar nas aplicações, melhorando a sua qualidade, pode permitir reduzir custos na perspetiva da manutenção do sistema.

1.5 Organização do documento

Este documento está dividido nos seguintes capítulos: Contexto, Estado da arte, Business Application Modeling Language, Validação sintática de modelos, Evolução dos modelos e Conclusões.

No capítulo "Contexto" é realizado um enquadramento ao tema, sendo apresentados alguns aspetos importantes da plataforma myMIS para uma melhor compreensão do restante documento.

Em "Estado da arte" são apresentados conceitos considerados importantes para a compreensão dos problemas e das respetivas soluções encontradas.

De seguida, em "Business Application Modeling Language" é realizada a descrição textual da linguagem que é o tema central deste documento. Neste capítulo são explicados todos os conceitos que constituem a linguagem, bem como os relacionamentos entre os mesmos, para que seja completamente percetível o que é e para que serve cada um dos elementos da BAMoL. Toda esta explicação é feita com o auxílio de um cenário de exemplo, de forma a tornar mais fácil a compreensão dos conceitos.

Posteriormente, em "Validação sintática de modelos" é descrito o processo de validação das aplicações existentes na plataforma. Para além de demonstrado o processo de validação, é explicada a ferramenta criada para o efeito. É ainda, através de um atividade experimental, validado o bom funcionamento da ferramenta, utilizando uma aplicação existente atualmente em funcionamento na plataforma myMIS.

No capítulo "Evolução dos modelos" é descrito o problema que as alterações nas aplicações realizadas ao longo do tempo podem trazer às mesmas. Serão apresentadas as alterações que se poderão tornar problemáticas, bem como exposto o problema da informação que por alguma razão, não tem qualquer utilidade mas permanece nas aplicações criadas.

Por fim, em "Conclusões" é feito um balanço do trabalho realizado nesta tese, bem como apresentadas sugestões de trabalho a realizar futuramente, por forma a melhorar as soluções encontradas e descritas no presente documento.

2 Contexto [2]

O desenvolvimento desta tese tem como base uma plataforma orientada à gestão, denominada myMIS. Neste capítulo irá ser realizada uma breve introdução à plataforma, bem como descritos alguns princípios do desenvolvimento em arquiteturas orientadas a modelos, linguagens de domínio específico e a framework REA, cujos princípios são a base para este sistema.

O myMIS é uma solução para desenvolvimento e deployment ágeis de Sistemas de Informação de Gestão. Neste momento existem dois pontos distintos do sistema: a plataforma e a aplicação web para o utilizador final. Esta tese será focada essencialmente na plataforma.

A plataforma myMIS é uma plataforma sustentada por uma abordagem baseada em modelos. O desenvolvimento de software recorrendo a arquiteturas orientadas a modelos, diferencia-se dos modelos tradicionais pela sua elevada capacidade de reutilização. Através da utilização de um desenho mais genérico, conseguem-se desenvolver modelos que dão resposta a uma grande diversidade de problemas, sem um aumento exponencial na complexidade do código desenvolvido.

As vantagens das arquiteturas orientadas a modelos não se concentram apenas no desenvolvimento do código, mas também nos modelos propriamente ditos. Ao contrário das aplicações desenvolvidas da forma tradicional, nestas existe uma maior separação entre o código e o domínio, permitindo a pessoas com conhecimentos básicos de informática e fortes conhecimentos do domínio, desenvolver o sistema conforme pretenderem [Selic, B., 2003].

Como cada cliente desta plataforma terá problemas e necessidades diferentes, isso fará com que existam múltiplos modelos no sistema. Para ser possível responder a todas estas soluções, é necessário que estas respeitem determinadas regras, obtidas através da utilização de uma linguagem de domínio específico.

[2] Realizado em conjunto com Diogo Teixeira

As linguagens de domínio específico são, como o nome indica, concebidas para serem utilizadas num domínio de problema em particular. Ou seja, permitem descrever aplicações que são interpretadas pelo domínio a que se referem. Neste caso em particular, a linguagem utilizada é a BAMoL.

A BAMoL é interpretada pelo runtime da plataforma myMIS e tem como objetivo permitir o desenvolvimento do mais variado tipo de soluções, através dos já referidos modelos.

Esta linguagem baseia-se na framework REA (Resources, Events, Agents) [McCarthy, W., 1982], estendendo os seus conceitos base: Recursos Económicos, Agentes Económicos, Eventos Económicos, Compromissos e Contratos. Com estes conceitos é possível descrever todas as entidades existentes para responder a um determinado problema e a forma como elas se relacionam, interagem e comportam.

A framework REA é orientada ao desenvolvimento de sistemas de contabilidade, centrando-se no registo dos diversos eventos económicos de uma organização. Assim, foi possível desenvolver um subsistema de contabilidade, que permite a criação e movimentação de contas, que se referem, por exemplo, aos agentes ou aos recursos, sendo os movimentos de um determinado recurso (não necessariamente monetário) despoletados pelos Eventos e Compromissos das transações.

Pode então resumir-se o funcionamento básico da plataforma myMIS da seguinte forma: a plataforma interpreta a linguagem BAMoL no formato de modelos, em que cada um representa uma solução para um determinado problema. Por sua vez, a BAMoL baseia-se na framework REA, permitindo assim que esta linguagem seja capaz de descrever um variado número de sistemas de informação de gestão.

2.1 A linguagem BAMoL

A Business Application Modeling Language é uma linguagem de domínio específico, baseada na teoria económica e na teoria da contabilidade e utilizada para possibilitar a implementação de soluções no domínio dos sistemas de informação para a gestão.

Com a BAMoL é possível descrever um modelo, que deve cumprir as variadas regras definidas na linguagem e, posteriormente, esse mesmo modelo em conjunto com a plataforma myMIS, permitem criar aplicações nas mais variadas áreas de negócio.

Esta linguagem tem sido utilizada até então sem qualquer formalização, sendo esse um dos problemas que esta tese pretende resolver. Essa carência de formalização dificulta tanto o desenvolvimento de modelos como a explicação da linguagem a alguém que se encontre fora da equipa de desenvolvimento da plataforma.

Para além disso, atualmente, a forma de guardar modelos localmente é realizada através de um conjunto de ficheiros que são impossíveis de ler por parte de qualquer pessoa. Com o trabalho desta tese pretende-se também eliminar esta complexidade, fazendo com que seja possível

guardar cada modelo em apenas um ficheiro, legível a qualquer desenvolvedor de aplicações BAMoL.

2.2 Organização do sistema

A organização da plataforma myMIS tem dois conceitos importantes a reter: o Tenant e o Template. É através destes dois elementos do sistema que é possível perceber como o mesmo se organiza.

Um Template é uma proposta de um modelo desenvolvido previamente, de forma a dar uma resposta a um determinado problema. Contém todos os elementos considerados necessários para criar uma solução.

Um Tenant pode ser definido como a instanciação de um modelo no sistema para o utilizador final e pode surgir através de um Template ou então, através de um desenvolvimento de raiz.

Para exemplo, consideremos dois clientes que necessitam de uma aplicação para gestão de tarefas, em alturas diferentes. Para o primeiro, é desenvolvido um modelo com as necessidades do cliente. Em relação ao segundo, pode ser proposto um modelo inicial, resultante da análise efetuado para o primeiro cliente. Ou seja, para o primeiro é desenvolvido um modelo de raiz, enquanto para o segundo, o modelo utilizado parte de um Template, que pode posteriormente ser alterado, de forma a adaptar-se às necessidades adicionais do cliente.

A figura seguinte mostra um esquema que representa a organização dos Tenants na plataforma, em que é possível que estes sejam baseados num Template ou, pelo contrário, desenvolvidos de raiz.

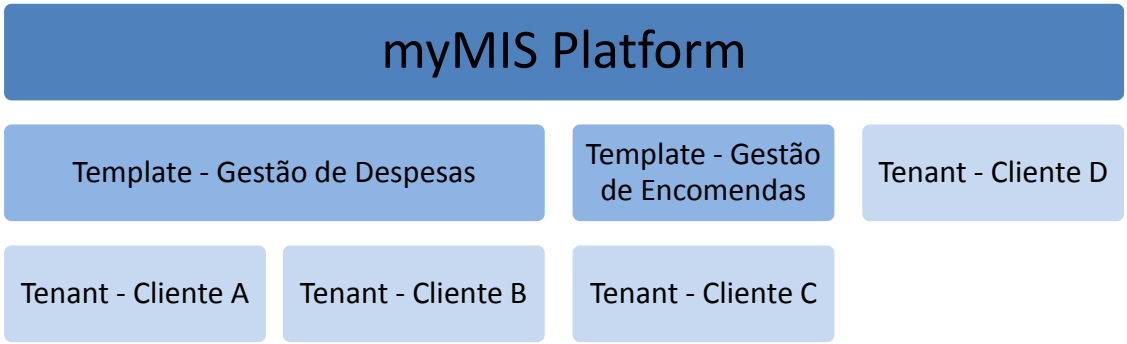


Figura 1 - Aprovisionamento de Contas

2.3 Criação de modelos

A criação de um modelo é o ponto inicial do desenvolvimento de uma solução na plataforma myMIS. Cada modelo (em conjunto com a plataforma) dá origem a uma aplicação diferente.

Existem duas formas possíveis para se proceder à criação de um modelo: a partir da utilização de um Template ou através de um desenvolvimento de raiz.

2.3.1 Utilização de Templates

Neste processo, o modelo é preenchido com os dados existentes num determinado Template. Desta forma, o modelo é inicializado já com alguns elementos criados, o que permite de forma mais rápida obter a solução pretendida para o problema em questão.

Apesar de esta ser uma forma mais simples, para quem desenvolve os modelos, de obter uma solução, para a equipa de desenvolvimento da plataforma myMIS acresce uma maior complexidade. Isto acontece porque é necessário garantir o correto funcionamento dos Templates quando a plataforma sofre alterações e também devido ao facto de, atualmente, a atualização dos Templates ser um processo complexo.

2.3.2 Desenvolvimento de raiz

Utilizando o desenvolvimento de raiz, em que o modelo se encontra inicialmente vazio, é necessário seguir algumas regras, para que o processo de desenvolvimento ocorra de forma correta.

Na figura que se segue é mostrada (e explicada no parágrafo seguinte) a ordem pela qual os elementos devem ser criados no modelo, para que o desenvolvimento deste seja realizado da forma mais correta.

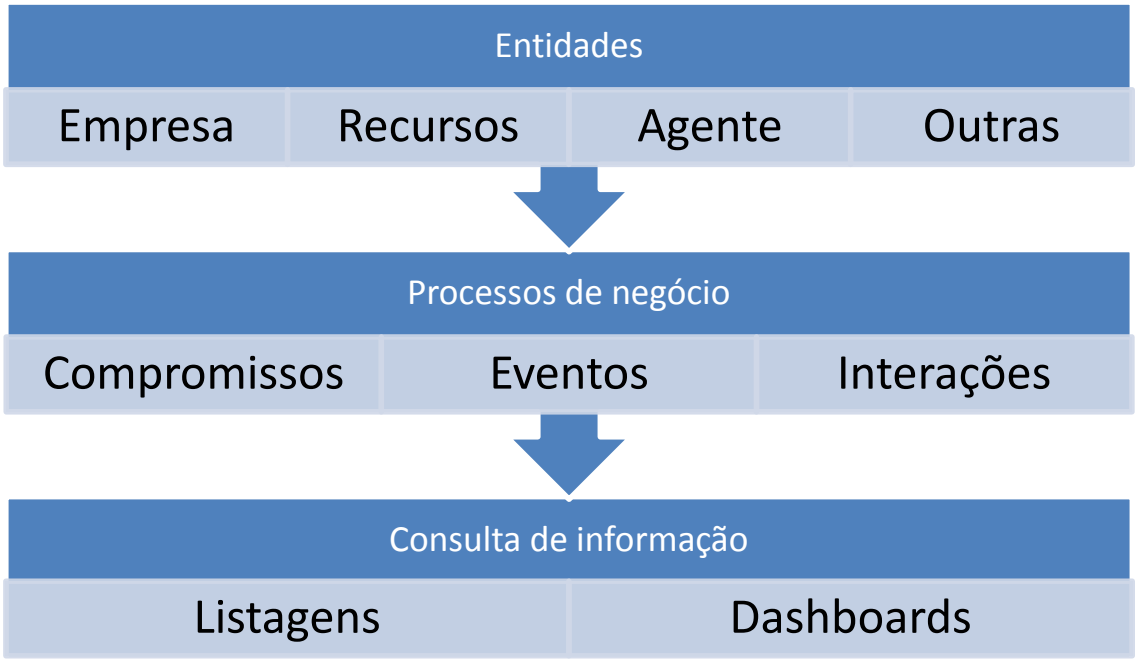


Figura 2 - Desenvolvimento de raiz de modelos

O desenvolvimento de raiz de um modelo deve seguir uma determinada ordem (mostrada na figura anterior), para que aconteça de uma forma mais simples, no sentido em que no momento de criação de um elemento, já exista no modelo toda a informação necessária para tal.

Assim sendo, os primeiros elementos a desenvolver no modelo devem ser os tipos correspondentes às Entidades. Sendo a plataforma myMIS direcionada aos sistemas de informação para a gestão, as operações são maioritariamente realizadas do ponto de vista da empresa que utiliza a plataforma como cliente final e, neste sentido, um destes tipos a ser criado em primeiro lugar deve ser um que seja representativo do conceito Empresa.

Após este primeiro passo, o passo seguinte deverá ser a definição dos tipos representativos dos Compromissos, Eventos e Interações. Com estes conceitos passa a ser possível descrever as transações existentes nas aplicações, utilizando os elementos do modelo criados no passo anterior.

Por fim, de forma a ser possível obter alguma informação útil do ponto de vista da análise para o gestor, podem ser desenvolvidos Dashboards e Listagens. Estes permitem ver a informação inserida utilizando os elementos criados no passo anterior a este, de uma forma agrupada e utilizando diferentes elementos (gráficos e tabelas) para consultar os dados pretendidos.

2.3.3 Ferramenta de desenvolvimento

De forma a facilitar o desenvolvimento de modelos na plataforma myMIS, existe uma ferramenta com uma interface web que o permite fazer e que tem como objetivos simplificar e agilizar o processo de criação/atualização de aplicações utilizando a linguagem BAMoL.

Com esta ferramenta é possível desenvolver modelos prontos a serem utilizados em apenas alguns minutos, pois a criação e atualização das aplicações ficam disponíveis para utilização de forma automática e imediata.

Nas figuras seguintes é possível observar alguns dos ecrãs desta ferramenta.

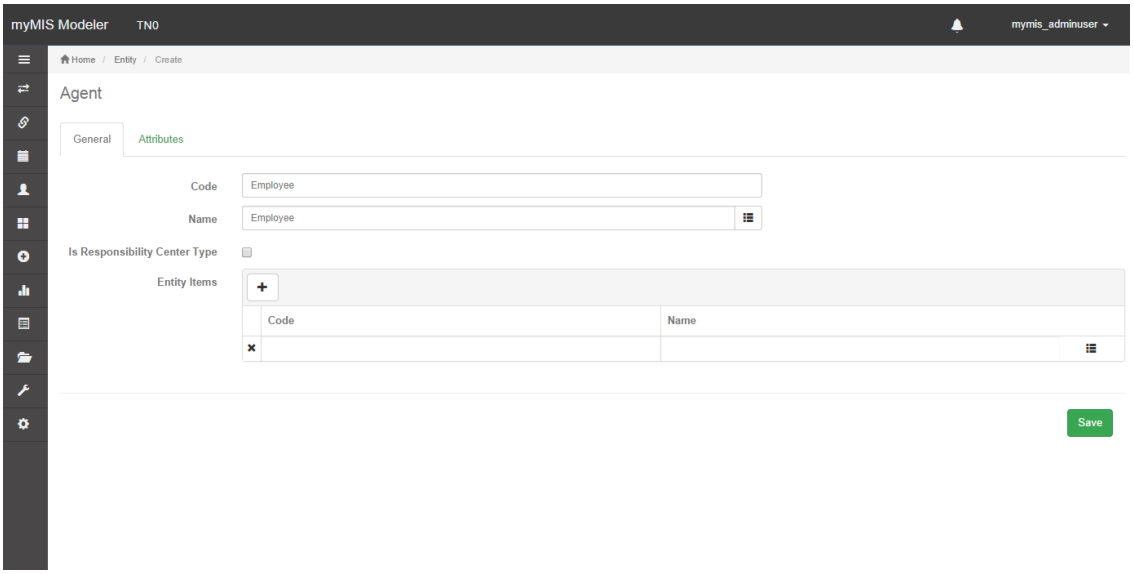


Figura 3 - Ferramenta de desenvolvimento: Criação de tipo de entidade

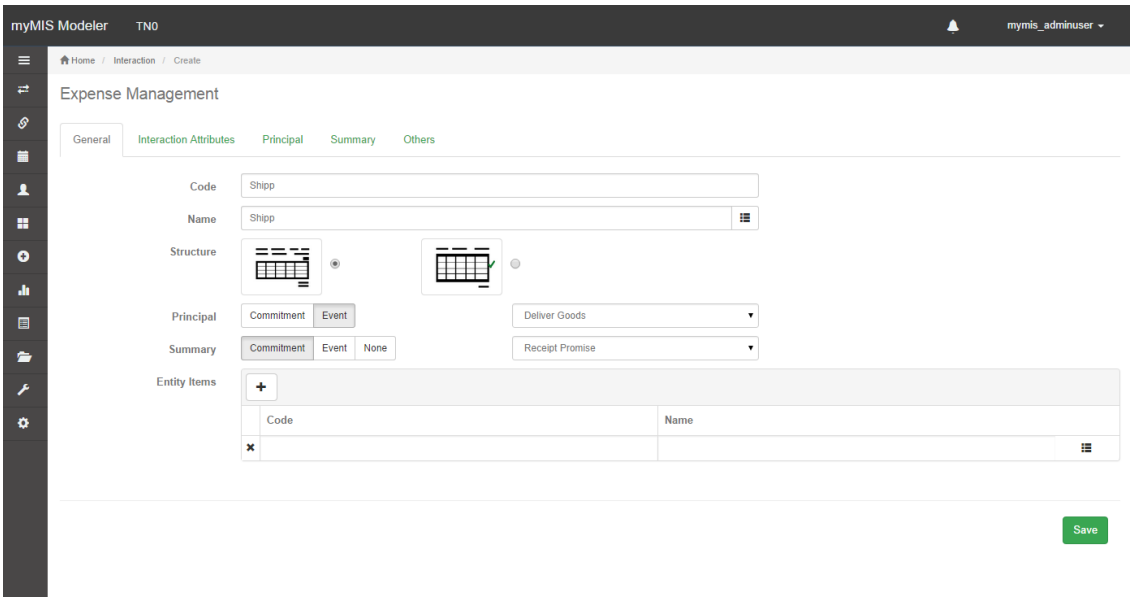


Figura 4 - Ferramenta de desenvolvimento: Criação de tipo de interação

As duas figuras anteriores representam os ecrãs onde é possível configurar os dois elementos mais importantes do sistema. Através destes formulários o desenvolvedor dos modelos é capaz de configurar toda a informação necessária para que uma aplicação funcione corretamente.

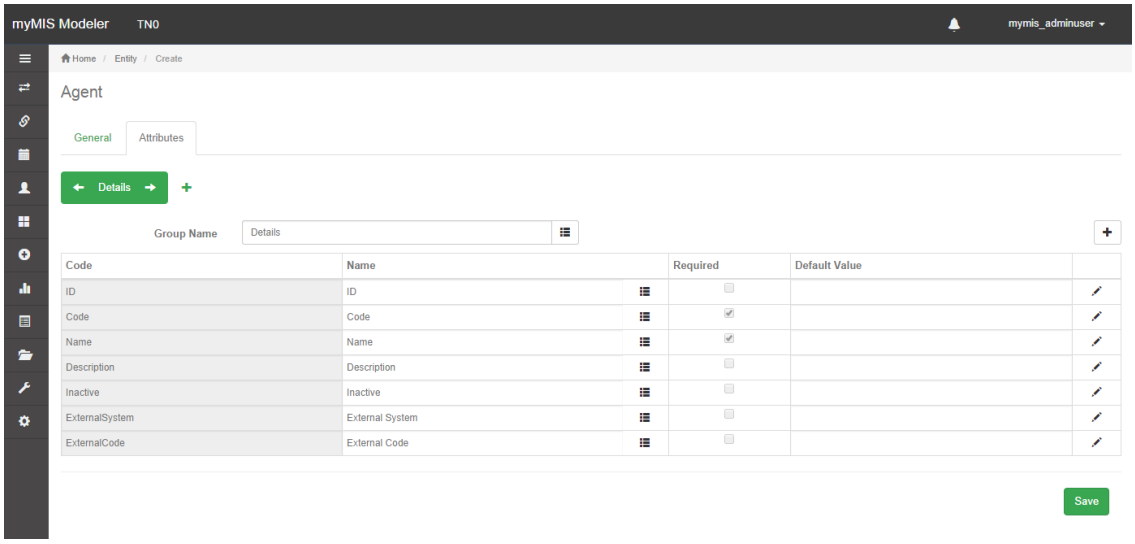


Figura 5 - Ferramenta de desenvolvimento: Listagem de atributos

Na figura anterior é possível visualizar uma listagem de atributos correspondeste a um tipo de entidade do sistema. Esta listagem permite editar os atributos já existentes e adicionar novos e encontra-se disponível tanto na criação de tipos de entidades como de tipos de interações.

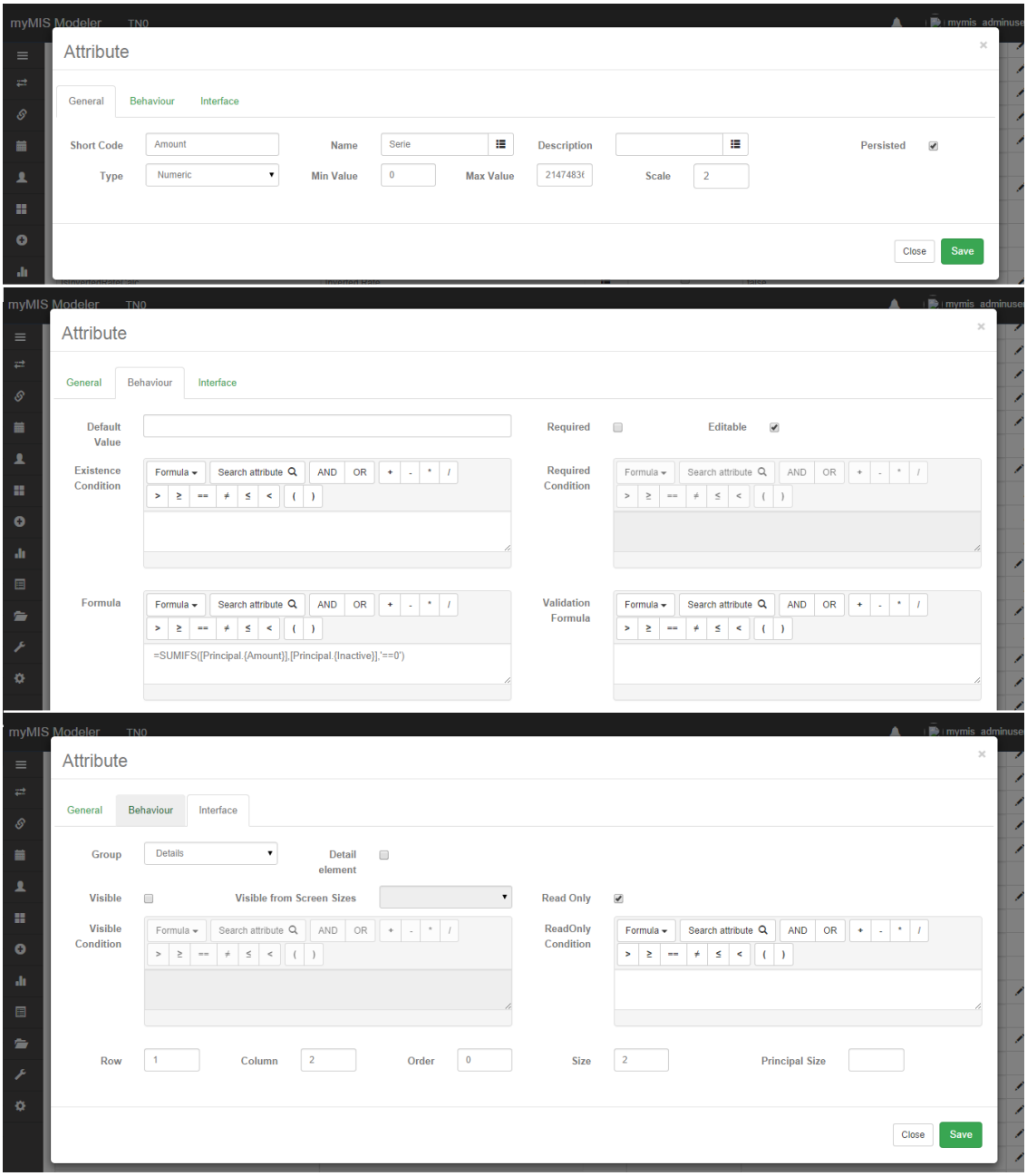


Figura 6 - Ferramenta de desenvolvimento: Criação de atributo

A configuração de cada um dos atributos existentes no modelo é realizada de acordo com o formulário apresentado na figura anterior. É assim possível configurar toda a informação necessária: as informações básicas (código, tipo de dados, etc.), dados relativos ao comportamento (fórmulas e valor por defeito, por exemplo) e propriedades relacionadas com a lógica de apresentação (visibilidade, localização no formulário, entre outras).

Atualmente, as grandes limitações desta ferramenta centram-se na validação sintática dos modelos criados e na dificuldade de exportação/importação de modelos. O primeiro destes dois pontos será estudado neste documento, enquanto o segundo se encontra em análise de forma a ser integrado brevemente nas funcionalidades disponibilizadas por esta ferramenta.

2.4 Validação de modelos

Neste momento, as únicas validações efetuadas acontecem no momento da gravação das entidades do modelo na estrutura de persistência existente. Essas validações são realizadas pelo mecanismo de persistência de dados da plataforma e apenas é validada a existência de todas as propriedades obrigatórias e o seu respetivo tipo de dados. Todas as outras validações necessárias para a avaliação do modelo do ponto de vista da sintaxe não são neste momento realizadas.

Assim sendo, a validação dos modelos da plataforma myMIS será um dos assuntos a tratar neste documento, de forma a permitir trazer um maior controlo dos modelos desenvolvidos à equipa de desenvolvimento.

3 Estado da arte

3.1 Linguagens de Domínio Específico

As linguagens de domínio específico (do inglês "domain specific languages", daqui para a frente referenciadas por DSL) têm um âmbito mais restrito, em comparação com as linguagens genéricas (como C# ou Java). Isto é, as linguagens que se enquadram dentro desta abordagem permitem o desenvolvimento de soluções apenas dentro do domínio em que se inserem.

Como exemplo de DSL existentes temos HTML, PERL, SQL e Prolog [Hudak, P., 1997].

Por não existir uma definição concreta sobre este conceito, podemos descrevê-lo recorrendo às seguintes definições:

"A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." [Deursen, A. *et al*., 2000]

"A computer programming language of limited expressiveness focused on a particular domain." [Fowler, M, 2010]

Partindo destas definições anteriores, podemos verificar que em ambas é referenciada a focalização destas linguagens num domínio restrito do problema. Esta é a particularidade que permite distinguir esta abordagem da outra, das linguagens com propósito genérico.

Comparando as duas abordagens, podemos enumerar como vantagens das soluções desenvolvidas com uma DSL a concisão, a simplicidade de manutenção das mesmas e a possibilidade de serem desenvolvidas por especialistas do problema (em vez de programadores) [Deursen, A. *et al*., 2000; Hudak, P., 1997]. Por outro lado, a utilização de DSL traz custos acrescidos de desenvolvimento e manutenção da linguagem, bem como aumenta a dificuldade de compreensão dos desenvolvedores de soluções [Deursen, A. *et al*., 2000].

As DSL podem ser divididas em três categorias: internas, externas e language workbenches [Fowler, M, 2010], explicadas de seguida.

Linguagens internas: Tendo uma linguagem de propósito genérico como base, uma DSL interna utiliza as funcionalidades dessa linguagem, estendendo-as de forma a criar as suas próprias especificidades. A DSL deve abstrair os conceitos da linguagem que lhe serve de base, dando a noção de que se trata de uma nova linguagem.

Linguagens externas: Nesta categoria, as DSL criadas são independentes de qualquer outra linguagem, tendo as suas próprias sintaxe e estruturas, ou então utilizando as de uma outra qualquer linguagem existente.

Language workbenches: Estas DSL são IDE's (Integrated Development Environment) utilizados para criar e definir as estruturas das linguagens de domínio específico, bem como soluções para essas mesmas linguagens.

3.2 Desenvolvimento orientado a modelos

Ao contrário do tradicional desenvolvimento de código numa linguagem de programação (como são o C# ou o Java), no desenvolvimento orientado a modelos (do inglês model-driven development) as aplicações são representadas através de modelos. Estes descrevem as soluções numa forma mais aproximada da linguagem do domínio do problema, permitindo assim um distanciamento dos conceitos tecnológicos. Isto torna possível que os peritos/especialistas no problema consigam estar mais próximos do processo de implementação da solução [Selic, B., 2003].

Um modelo pode ser definido como um conjunto de elementos, que se podem relacionar e interagir, de forma a descrever uma solução para um determinado problema. Esses elementos respeitam uma determinada formalidade, que é definida através de um meta-modelo.

O meta-modelo define o conjunto de conceitos e regras que devem ser respeitados para construir os modelos, sendo possível representá-lo através de uma domain-specific language. A DSL "engloba o meta-modelo, bem como a sintaxe que permite definir os modelos" [Voelter, M. *et al*.].

3.3 REA Accounting Model

REA – Resources, Events and Agents – é uma framework de contabilidade "desenhada para ser utilizada em ambientes de dados partilhados, em que contabilistas e não contabilistas pretendem manter informação sobre o mesmo conjunto de acontecimentos" [McCarthy, W., 1982]. Surgiu como forma de ser uma alternativa válida ao tradicional modelo contabilístico de dupla-entrada.

Como o seu nome indica, três dos conceitos fundamentais deste modelo são Recursos (Resources), Eventos (Events) e Agentes (Agents). A estes acrescem ainda Compromissos (Commitments) e Contratos (Contracts). Estes conceitos podem definir-se de uma forma simples, da seguinte maneira [McCarthy, W., 1982; Hruby, P. *et al*.]:

- *Recurso*: Qualquer objeto controlado por uma empresa, com valor económico e que pode ser transacionado (exemplo: materiais);
- *Evento*: Ação que prevê a movimentação de recursos (exemplo: venda de produtos). Pode ser um incremento ou um decremento da quantidade de um determinado recurso;
- *Agente*: Entidade que participa nos eventos e compromissos (exemplo: clientes);
- *Compromisso*: É a obrigação que um agente tem para com outro, de realizar um evento no futuro (exemplo: compromisso de pagamento a 30 dias, no caso de uma compra de produtos). A forma de cumprir um compromisso (fulfillment) pode ser feita de duas maneiras: através de um novo compromisso ou de um evento;
- *Contrato*: Conjunto de compromissos e de condições. Essas mesmas condições, quando se verificam, podem criar compromissos adicionais.

Com o esquema seguinte [Hruby, P. *et al*.] é possível perceber de forma mais clara como os conceitos fundamentais deste modelo de contabilidade se relacionam.

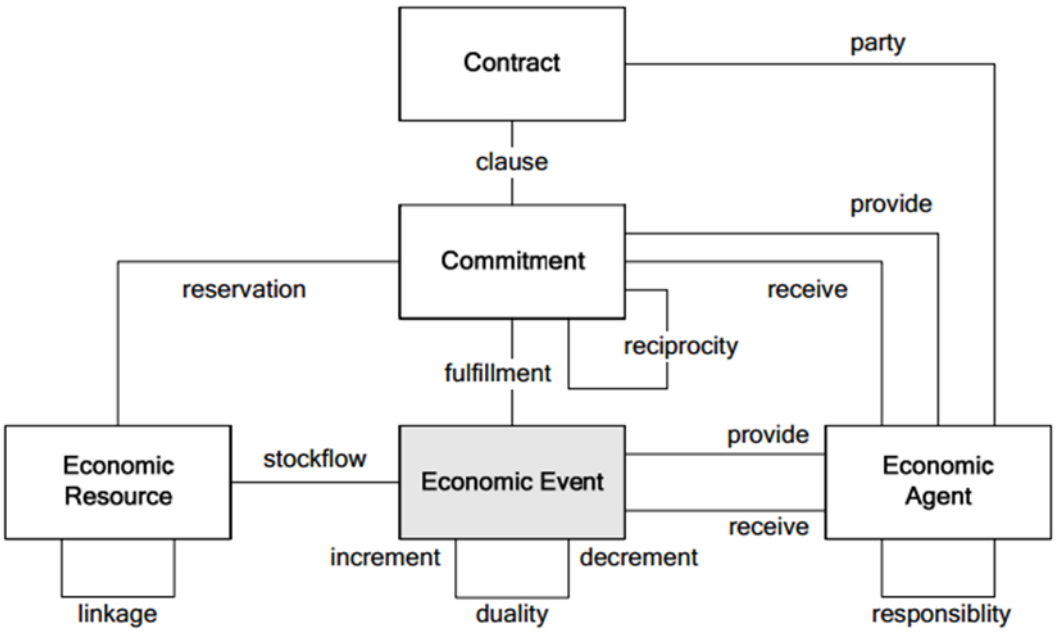


Figura 7 - REA Accounting Model: Conceitos fundamentais

Com esta estrutura, é possível guardar informação sobre tudo o que aconteceu num sistema, permitindo saber o que deu origem a quê, garantindo assim a rastreabilidade de todos os acontecimentos que ocorreram no sistema.

3.4 Adaptive Object-Model

O AOM é um estilo de arquitetura que permite construir sistemas de uma forma distinta e que difere em vários aspetos das abordagens mais convencionais, como é o caso dos sistemas orientados a objetos.

Estes sistemas seguem uma abordagem alternativa, no sentido em que o modelo de negócio é guardado na base de dados [Yoder, J. *et al*. (2)], em vez das tradicionais classes. Todas as entidades necessárias para dar resposta a um problema são descritas, bem como os seus comportamentos, dando assim origem a meta-dados.

Para além da descrição das entidades do negócio, também as relações entre todos os constituintes do sistema são descritas através de meta-dados, que em runtime são interpretados [Yoder, J. *et al*. (1)].

Esta abordagem permite criar sistemas mais flexíveis, no sentido em que qualquer alteração nas entidades ou regras de negócio pode não obrigar a uma alteração do código, mas apenas a uma alteração dos meta-dados que representam o sistema. Outro ponto importante é o facto de, através desta abordagem, não ser necessário saber programar para se proceder a alterações do sistema: existindo uma forma de alterar os meta-dados, um especialista no problema representado pelo sistema pode efetuar alterações, sem escrever uma linha de código.

Arquitetura AOM

Os Adaptive Object-Models fazem uso de diversos padrões, de forma a permitir construir toda a sua arquitetura [Yoder, J. *et al*. (2)]. Para a compreensão do tema que este documento trata, é necessário compreender o padrão TypeObject. Este permite substituir a tradicional estrutura orientada a objetos, em que cada classe representa um objeto diferente, podendo existir a necessidade de criar várias subclasses de uma determinada abstração do sistema.

Assim, este padrão torna possível substituir todas essas subclasses por instâncias de um determinado tipo (que representa a abstração referida anteriormente) [Yoder, J. *et al*. (2)]. Desta forma, a adição de uma nova subclasse não necessita de novo código (e consequente compilação), mas apenas da descrição através de meta-dados.

Para ser possível uma melhor compreensão deste conceito, temos o seguinte exemplo:

- Sistema que contém uma abstração Veículo e as seguintes subclasses: Carro, Camião e Barco;
- Cada objeto no sistema é uma instância de Carro, Camião ou Barco.

Após a utilização do padrão TypeObject o sistema seria organizado da seguinte forma:

- TipoVeículo: representa a abstração Veículo;
- Várias instâncias de TipoVeículo (Carro, Camião e Barco), representadas através de meta-dados, em que cada objeto no sistema contém uma referência a TipoVeículo.

4 Business Application Modeling Language

Sobre a formalização de linguagens naturais, Susan Haack[3] referiu o seguinte: "Ao formalizar, procura-se generalizar, simplificar, e aumentar a precisão e o rigor". Apesar de esta frase não ter sido aplicada ao problema das linguagens computacionais, pode ser utilizada no caso da linguagem a que esta tese se refere.

A formalização da linguagem BAMoL tem um papel importante, pois neste momento não existe qualquer formalização da mesma, o que torna a criação de modelos mais complexa e a validação sintática impossível de realizar. Sem esta formalização da linguagem torna-se também muito difícil a explicação da linguagem a pessoas que estão a ter o primeiro contacto com a mesma.

Neste capítulo são apresentados os conceitos desta linguagem e as relações entre os mesmos, divididos em dois grupos: os relacionados com a lógica de negócio (meta-modelo da linguagem) e aqueles que permitem adicionar mais alguns comportamentos importantes para o desenvolvimento das aplicações.

Por fim, é também apresentada a gramática da BAMoL, criada a partir da descrição da linguagem e que será utilizada para a validação sintática de modelos.

4.1 Cenário de exemplo

Com o objetivo de ser mais fácil perceber os conceitos desta linguagem é apresentado de seguida um exemplo, que servirá de apoio a toda a explicação, que descreve um problema de gestão de encomendas e expedição das mesmas.

[3] Haack, S. (2002). Filosofia das Lógicas. São Paulo: Editora UNESP (ISBN: 9788571393998)

O objetivo deste sistema é permitir registar encomendas de produtos e, numa fase posterior, registar a expedição das mesmas.

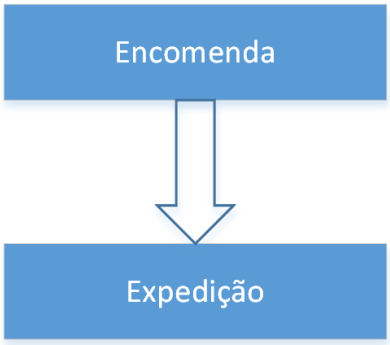


Figura 8 - Cenário de exemplo: fluxo de negócio

Para o primeiro momento (registo de encomendas) é necessário registar um conjunto de compromissos de entrega de produtos a um determinado cliente. O exemplo da inserção de dados deve ser semelhante ao da figura seguinte.

Cliente CLIENTE001

Produto	Quantidade	Preço Unitário (€)	Preço Total (€)
PRODUTO_01	1	150.00	150.00
PRODUTO_03	3	34.00	102.00
PRODUTO_34	10	4.50	45.00

Figura 9 - Cenário de exemplo: registo de encomenda

Partindo dos dados anteriores, obtemos o seguinte esquema que representa o relacionamento entre as entidades e os recursos envolvidos neste mecanismo. É a partir da figura seguinte que é possível perceber os dados necessários ao desenvolvimento de modelos na BAMoL.

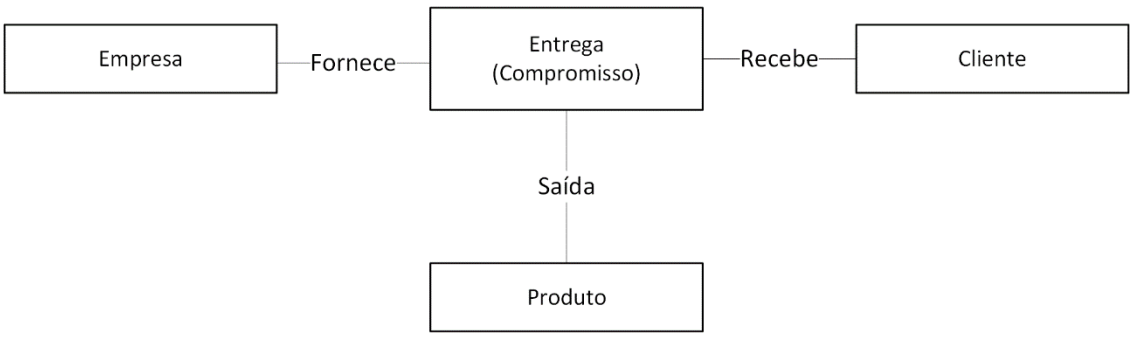


Figura 10 - Cenário de exemplo: representação de encomenda

Numa segunda fase é necessário efetuar o registo da expedição dos produtos encomendados anteriormente. Para tal é necessário registar o evento da entrega de produto ao cliente, que irá

satisfazer o compromisso de entrega de produto anterior (registado na encomenda). Além da entrega de produto, neste momento também acontece o pagamento dos produtos e a geração de compromissos de pagamento de um imposto, neste caso o IVA. A inserção de dados deve seguir o esquema representado na figura seguinte.

Cliente CLIENTE001 Importar encomendas

Produto	Quantidade	Preço Unitário (€)	Preço Total (€)
PRODUTO_01	1	150.00	150.00
PRODUTO_03	2	34.00	68.00
PRODUTO_34	5	4.50	22.50

IVA	A pagar (€)
23%	56.085

Total a pagar 240.50

Figura 11 - Cenário de exemplo: registo de expedição

Utilizando as informações anteriores relativas ao registo da expedição, temos as seguintes representações:

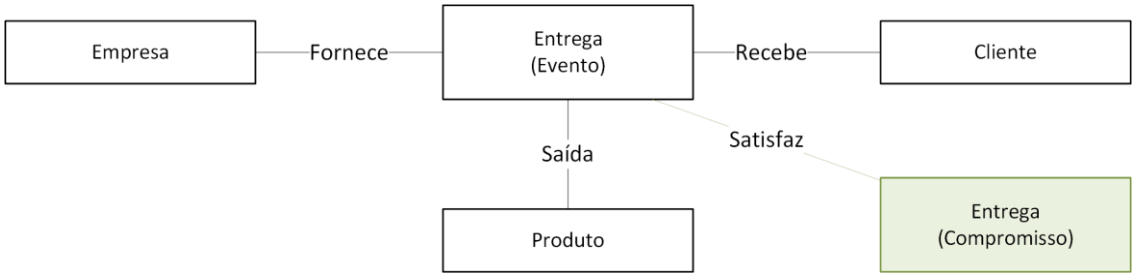


Figura 12 - Cenário de exemplo: representação de expedição

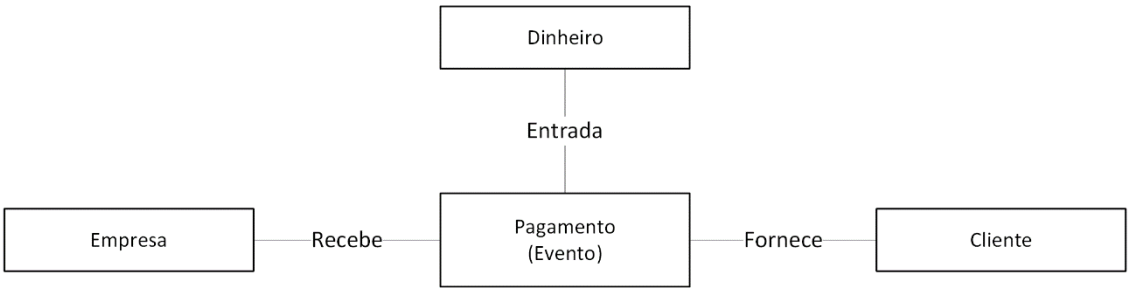


Figura 13 - Cenário de exemplo: representação de pagamento de expedição

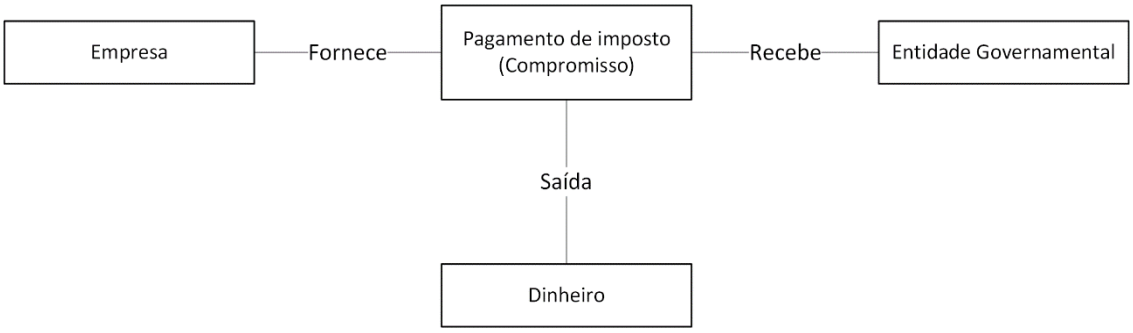


Figura 14 - Cenário de exemplo: representação de pagamento de imposto

A partir destas representações é possível descrever todo o sistema, desde as entidades necessárias até às interações entre as mesmas. Assim sendo, este exemplo será utilizado no subcapítulo seguinte, como apoio à explicação efetuada no mesmo.

4.2 Conceitos de negócio

Neste capítulo serão descritos os conceitos da linguagem que permitem descrever as regras de negócio da solução dos problemas. Para tal, de forma a ser mais inteligível, toda a explicação será feita com o apoio do exemplo apresentado anteriormente.

Em primeiro lugar, é necessário conhecer os conceitos base da linguagem. Para tal, é importante realizar uma conversão das representações apresentadas no exemplo para uma outra que tenha por base a REA. Assim sendo, utilizando o cenário da encomenda, obtemos a representação seguinte.

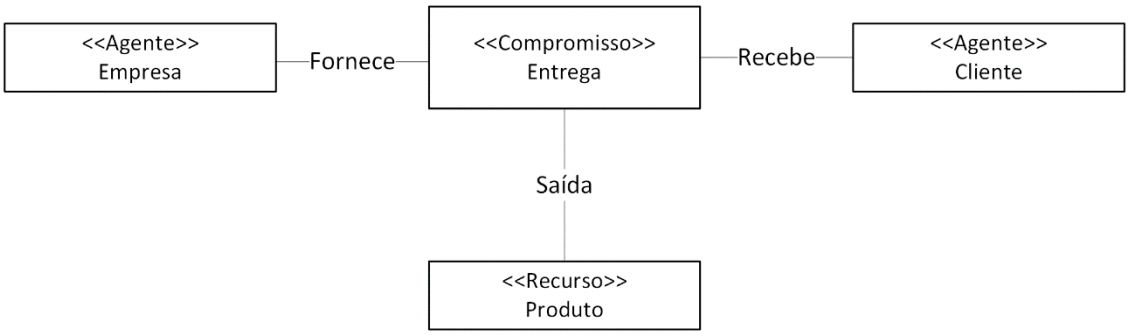


Figura 15 - Representação de mecanismo de encomendas utilizando conceitos REA

Partindo da figura anterior, podemos dizer que a realização de uma encomenda representa um conjunto de compromissos entre os dois agentes, em que um deles se compromete a entregar um recurso ao outro. O conjunto destes compromissos denomina-se por interação.

A partir deste simples exemplo, podemos extrair alguns conceitos importantes a reter: interação, compromisso, recurso e agente. Para além destes, existem ainda outros conceitos elementares, nomeadamente o evento e o processo. Tendo por base tudo isto, a linguagem

BAMoL contém elementos que permitem descrever uma grande variedade de soluções utilizando estas ideias e que são explicados em seguida.

4.2.1 Resource Type

Corresponde ao conceito "Recurso" da REA e tem como objetivo descrever todos os objetos controlados por uma empresa, com valor económico e que podem ser transacionados. São recursos no exemplo apresentado o Produto e o Dinheiro.

Cada entidade deste tipo, criada no modelo, é passível de ser considerado um centro de responsabilidade, de forma a ser possível aumentar a informação de gestão na aplicação. Também os elementos da linguagem Agent Type e User Defined Type têm esta propriedade.

4.2.2 Agent Type

Corresponde ao conceito "Agente" da REA e tem como objetivo descrever todas as entidades que podem ser parte de um transação de recursos, seja como recebedor ou como fornecedor dos mesmos. Neste conceito, são exemplos a Empresa, o Cliente e a Entidade Governamental.

Em cada modelo deve existir um, e só um, Agent Type que representa a empresa para o qual o sistema foi criado.

4.2.3 User Defined Type

Este conceito da BAMoL não tem qualquer correspondência na REA e pretende dar a possibilidade de serem descritas outras entidades que não sejam nem agentes nem recursos, mas que têm importância para os modelos criados. Desta forma, é possível adicionar informação adicional às transações existentes, de forma a aumentar a informação disponível no sistema.

Podem ser descritas com este conceito entidades, como por exemplo as classes de IVA, as formas de pagamento ou o veículo.

4.2.4 Event Type

Este conceito está relacionado diretamente com o conceito da REA "Evento" e descreve uma transação de um recurso que ocorreu num determinado momento entre dois agentes.

No evento, o controlo do recurso transacionado passa de um agente (o fornecedor) para outro (o recebedor). Utilizando o cenário de exemplo apresentado, temos o caso da Expedição, em que a Empresa transaciona o Produto para o Cliente, ficando este último com o controlo do recurso em questão.

As propriedades necessárias para a definição deste elemento da linguagem são descritas de seguida, aquando da explicação do conceito Commitment Type.

4.2.5 Commitment Type

Este conceito está relacionado diretamente com o conceito da REA "Compromisso" e descreve o compromisso de um agente para com um outro, da realização de um evento num momento futuro. Todos os compromissos existentes devem ser satisfeitos por um evento ou por outro compromisso que ocorre num momento posterior.

No cenário de exemplo apresentado, temos o caso da Encomenda, em que a Empresa se compromete a entregar o Produto ao Cliente num momento futuro. Este compromisso será satisfeito mais tarde, aquando da realização da Expedição.

Da definição deste elemento da linguagem (e também do conceito Event Type) faz parte um agente (Provider Agent) que irá perder os direitos sobre o recurso transacionado (Resource). Esses direitos passarão a pertencer a um outro agente (Receiver Agent). Na figura seguinte é possível ver a forma como estes conceitos se relacionam, tanto com o conceito Agent Type como com o Resource Type.

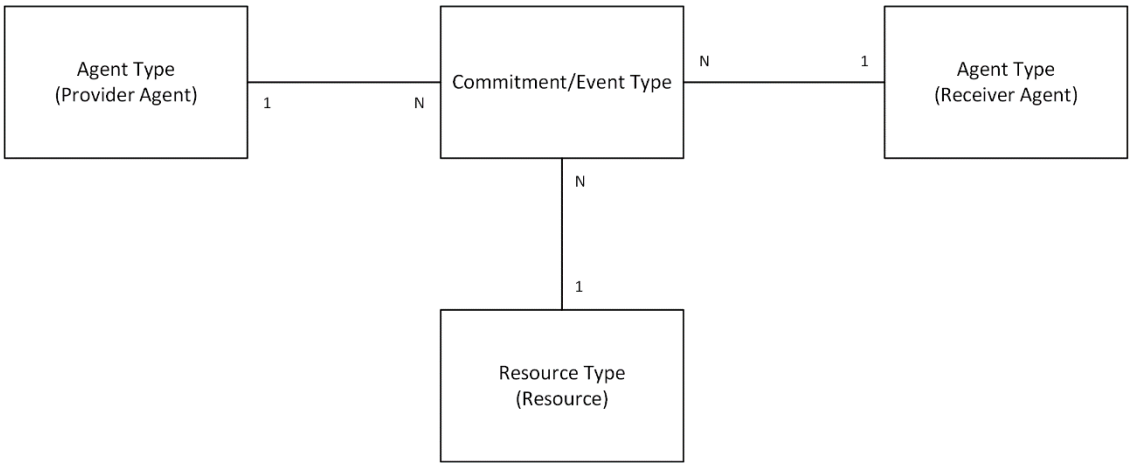


Figura 16 - Relação Event/Commitment Type com Agent e Resource Types

Observando a figura anterior, é possível perceber que os Commitment e Event Type têm sempre dois Agent Types e um Resource Type envolvidos: **Resource Type** representa qual o tipo de recurso a ser transacionado; **Provider Agent Type** representa qual o tipo de agente que irá perder direitos sobre o recurso, na perspetiva da empresa para a qual o modelo é descrito; **Receiver Agent Type** representa qual o tipo de agente que irá adquirir direitos sobre o recurso, na perspetiva da empresa para a qual o modelo é descrito.

Para além disso, estes dois conceitos partilham ainda mais duas propriedades: **Kind** e **Activity Kind**. A primeira tem apenas dois valores possíveis e descreve se o tipo de compromisso/evento vai dar origem a um incremento ou a um decremento do recurso transacionado, na perspetiva da empresa para o qual o modelo é criado. Em relação à segunda propriedade, esta tem três

valores possíveis (Operational, Financing e Investment) e representa qual o tipo de atividade no ponto de vista do cash flow (e é utilizada para ser possível obter posteriormente mapas de fluxos de caixa).

4.2.6 Interaction Type

Este conceito representa um momento em que ocorre a realização de um conjunto de compromissos e eventos, representando uma interação entre vários agentes e trocas de vários recursos. Fazendo a ligação com a REA, aproxima-se do conceito Contrato.

No cenário de exemplo temos duas interações: a Encomenda e a Expedição. Ambas representam um momento em que existe um conjunto de compromissos e/ou eventos a serem realizados entre vários agentes.

Para a descrição de Interaction Types há dois cenários possíveis:

1. Existem dois compromissos ou eventos, cuja relação entre eles é de N para 1. Este é o caso representado pela Expedição do cenário de exemplo.
   Neste caso, a relação entre os compromissos/eventos representará uma troca de recursos entre os tipos de entidades envolvidas (ambos são fornecedores e recebedores), em que um dos compromissos/eventos (Summary) resultará da agregação do outro (Details).
   Na Expedição, Details é representado pelo evento Entrega e Summary pelo evento Pagamento.
   Neste cenário existe a restrição de que os compromissos/eventos envolvidos têm de ser compatíveis. Isto significa que não podem ser do mesmo tipo e, para além disso, caso tenham a mesma natureza (ambos compromissos ou ambos eventos) o valor da propriedade Kind tem obrigatoriamente que ser diferente.

Figura 17 - Interaction Type: Relação entre compromissos/eventos

2. Existe apenas um Commitment/Event Type. É representado pela Encomenda, descrita no cenário de exemplo.
   Esta situação representa o caso em que apenas há um tipo de transação. Ou seja, dos tipos de entidades envolvidas apenas uma é fornecedora do recurso e outra apenas é recebedora do mesmo. Ou seja, nesta situação e em comparação com o cenário anterior, apenas é definida a propriedade Details.

Desta forma, utilizando os dois conceitos mencionados anteriormente, nomeadamente, Details e Summary, é possível descrever a estrutura base de uma interação, que tem por base as relações que podem ser observadas na imagem seguinte.

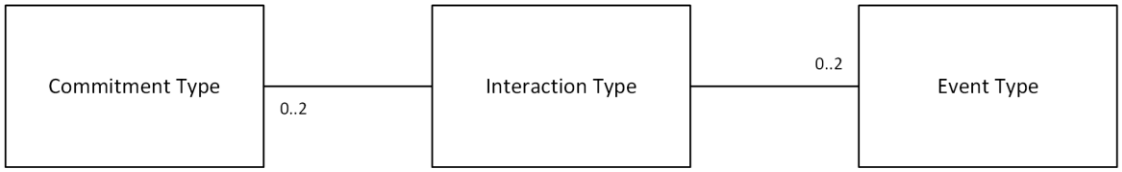
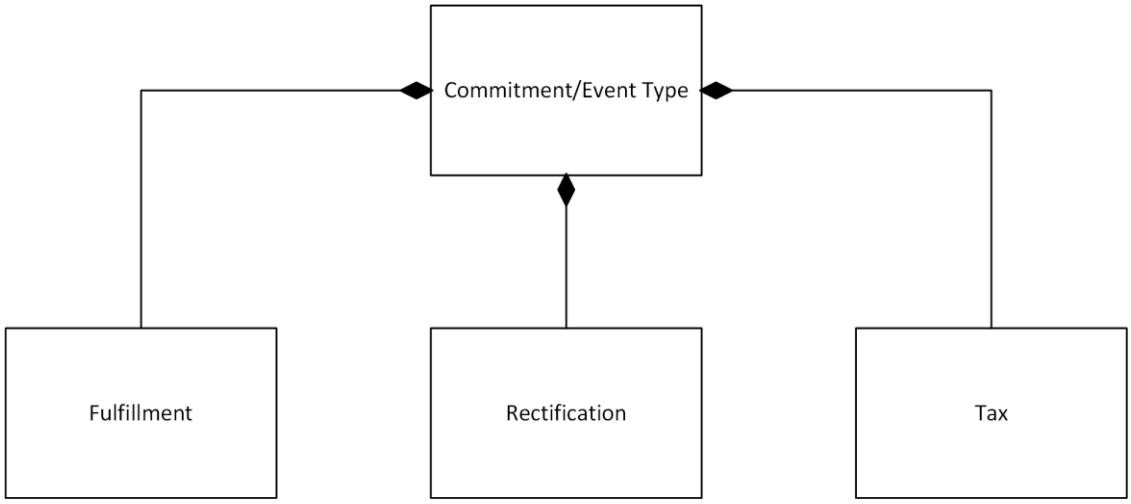


Figura 18 - Relação Interaction Type – Commitment/Event Types

É ainda possível descrever mais conceitos relativos ao Interaction Type, sendo eles: **Fulfillments, Rectifications e Taxes**, estando estes sempre relacionados com um Commitment ou Event Type presentes no Interaction Type, como é esquematizado na figura seguinte.



Os mecanismos relacionados com estes três conceitos serão apresentados de seguida.

Por último, é ainda possível indicar informações relativas ao serviço e-Fatura[4], nomeadamente se o Interaction Type corresponde a um documento assinado digitalmente ou a um documento de transporte de bens.

4.2.6.1 Mecanismos de Fulfillment e de Rectification

Os primeiros são utilizados quando se pretende descrever que o Commitment/Event Type em questão satisfaz um tipo de compromisso existente num outro Interaction Type. Desta forma, é possível descrever uma espécie de dependência entre interações.

[4] Serviço fornecido pelo Ministério das Finanças do Governo Português: http://info.portaldasfinancas.gov.pt/pt/faturas/sobre_efatura.html

Como podemos verificar no cenário de exemplo, na Expedição o evento Entrega resulta da satisfação do compromisso Entrega que faz parte da Encomenda. Assim sendo, caso não existisse qualquer Encomenda realizada, não poderia ser realizada qualquer Expedição.

No caso dos Rectifications, o objetivo é descrever a forma como as instâncias dos seus Commitment/Event Types podem servir de retificação a algo (instâncias de Commitment/Event Types) que foi realizado num outro Interaction Type. Como exemplo, temos as Notas de Crédito, que num cenário de um sistema de faturação poderiam ser utilizadas como forma de anular uma Fatura já emitida anteriormente.

A descrição destes dois conceitos é feita de acordo com as mesmas propriedades, enunciadas de seguida.

A propriedade **Kind** representa o tipo de comparação a efetuar de forma a avaliar se o compromisso pode ser considerado fechado (quando está satisfeito/retificado totalmente) e pode ter um de quatro valores diferentes:

- *Equal*: O compromisso a satisfazer/retificar só será dado como fechado se o valor do compromisso/evento que o satisfaz/retifica tiver exatamente o mesmo valor;
- *Less than:* O compromisso/evento que satisfaz/retifica pode ter um valor inferior ao compromisso a satisfazer/retificar;
- *Greater Then:* O compromisso/evento que satisfaz/retifica pode ter um valor superior ao compromisso a satisfazer/retificar;
- *Any:* O compromisso será considerado fechado por qualquer valor que tenha o compromisso/evento que o satisfaz/retifica.

De forma a ser possível conhecer qual o agente ao qual o compromisso/evento está atribuído é necessário definir a propriedade **AgentAttribute**, que representa um atributo do Interaction Type. É também possível definir um conjunto de condições (definidas em **Conditions**) que serão utilizadas como filtro para selecionar os dados corretamente.

De forma a ser possível satisfazer/retificar um compromisso é necessário existir um outro compromisso/evento que representa essa situação. Para isso, é necessário definir um mapeamento entre os atributos do compromisso a satisfazer/retificar e os do compromisso/evento que satisfaz/retifica o primeiro. Isso é feito recorrendo à propriedade **Attributes**, em que é descrita uma lista de pares de atributos, em que cada um descreve a correspondência entre um atributo do compromisso a satisfazer/retificar e um atributo do compromisso/evento do Interaction Type atual que irá satisfazer/retificar o primeiro.

Por último é possível definir, com a propriedade **List**, qual a listagem que irá ser utilizada para mostrar as instâncias dos Commitment/Event Types, possíveis de utilizar no contexto em que se inclui o Fulfillment/Rectification.

4.2.6.2 Mecanismo de Taxes

Este conceito permite descrever um conjunto de impostos a serem aplicados ao Interaction Type em questão, calculados sempre a partir do tipo de compromisso/evento com o qual se relacionam.

Temos como exemplo, o tipo de compromisso Pagamento de Imposto, que consta no cenário de exemplo apresentado. Os valores dos compromissos deste tipo serão sempre obtidos a partir dos valores dos eventos do tipo Entrega. Desta forma, é possível gerar de forma automática os compromissos/eventos de pagamento de impostos que resultam da realização de transações no sistema.

Para descrever os elementos deste mecanismo de impostos é necessário definir qual o Commitment/Event Type que representa o evento realizado ou compromisso a executar do pagamento do imposto em questão, através da propriedade **ResultType**, bem com o User Defined Type que representa qual o tipo de imposto em questão, utilizando a propriedade **EntityType**.

Para além das propriedades anteriores é necessário definir uma lista de correspondência entre atributos, que permite saber aqueles que contêm informação relevante para o cálculo do imposto, feito através da propriedade **Attributes**.

4.2.6.3 Lógica de apresentação

Relativamente à disposição dos elementos no ecrã, os atributos do Interaction Type irão ser colocados sempre no início da página, seguidos de uma grelha que irá conter um dos (ou o único) Commitment/Event Types escolhidos. No caso de serem dois os escolhidos, deve ser indicado qual o que irá ser representado por uma grelha (através da propriedade Details). O outro (indicado pela propriedade Summary) será colocado depois dessa mesma grelha, situado à direita. Caso existam Taxes, estes serão mostrados alinhados com este último Commitment/Event Type, mas situados à esquerda. Esta organização pode ser vista com o exemplo da figura seguinte.

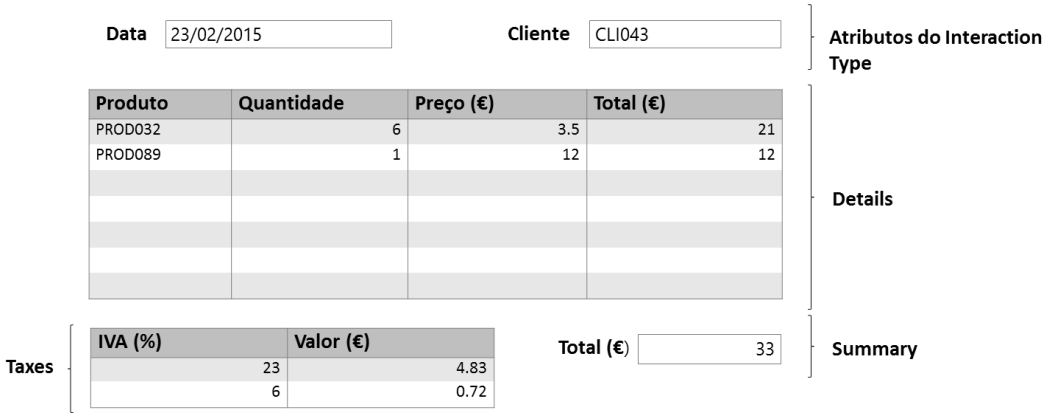


Figura 19 - Interaction Type: Estrutura do ecrã

Ainda relativo à estrutura visual de cada Interaction Type, deve ser indicado o tipo de ecrã desejado, através da propriedade Pattern. Existem dois valores possíveis, nomeadamente AddToDetails e SelectInDetails, sendo que o primeiro representa que é possível adicionar novos registos de um Commitment/Event Type (Details) e o segundo que é apenas possível selecionar registos já existentes.

4.2.7 Process Type

Este conceito representa um bloco de operações (Interaction Types) que se relacionam e que, em conjunto, descrevem os passos de um processo de negócio do ponto de vista da empresa.

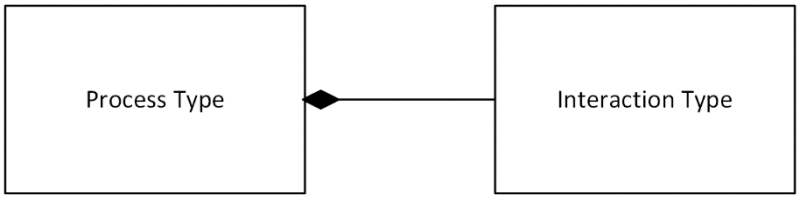


Figura 20 - Relação Process Type - Interaction Type

No cenário de exemplo apenas temos um tipo de processo, que resulta do conjunto dos dois tipos de interações existentes. Este conjunto representa um processo de negócio, no sentido em que a Encomenda e a Expedição se relacionam e se encontram no seguimento uma da outra.

É possível no mesmo sistema existirem vários tipos de processos de negócio. Por exemplo, o mesmo sistema pode ter o tipo de processo de Vendas e o de Compras a coexistirem sem qualquer problema.

4.2.8 Entity Item Type

Este conceito surge para suprir a necessidade de ser necessário em alguns Types descrever estruturas complexas (com vários atributos) com múltiplas instâncias. Apenas nos casos dos Agent, Resource, User Defined e Interaction Types é possível definir Entity Item Types.

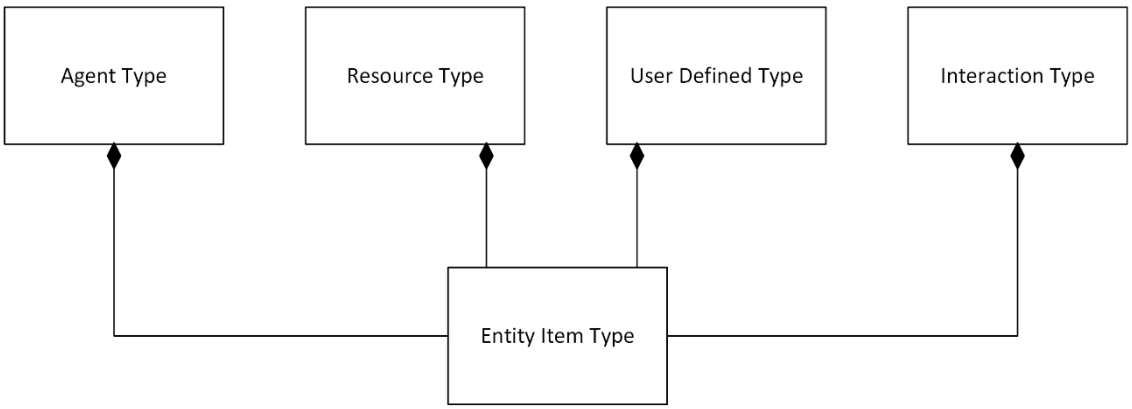


Figura 21 - Relações de Entity Item Type

Como exemplo da aplicação deste conceito na implementação de uma solução, temos um caso em que existe a necessidade de o recurso Produto ter uma lista de valores para o seu preço consoante o país. Neste caso, é necessário uma estrutura que permita para uma entidade do modelo guardar uma lista de conjuntos de valores (preço e país). Isso apenas é possível recorrendo a este conceito.

4.2.9 Attribute

Com os conceitos anteriores é possível descrever quais as entidades do modelo, mas para se definirem todas as propriedades que uma entidade contém é necessário recorrer a este conceito.

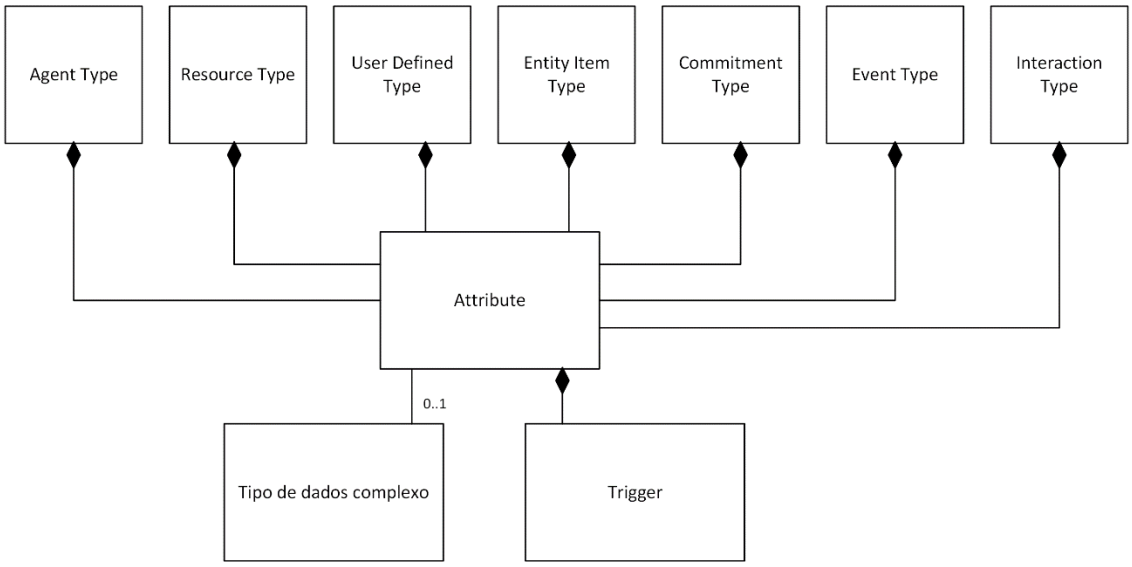


Figura 22 - Relações de Atributo

Com o Attribute é possível descrever o conjunto de atributos e, assim, representar um grande número de comportamentos diferentes. Para representar esses mesmos comportamentos é necessário conhecer todas as propriedades que são possíveis de descrever e que serão enumeradas e explicadas em seguida.

As propriedades podem ser divididas em dois grupos: um que representa as propriedades necessárias à criação da lógica de negócio; outro que descreve a organização em termos de apresentação.

4.2.9.1 Lógica de negócio

Para descrever um atributo devemos indicar qual o seu tipo de dados (**Type**), cujos valores possíveis podem ser divididos em duas categorias:

- Base: texto, texto protegido, número inteiro, número decimal, data, booleano e ficheiro;

- Complexo: representa um atributo cujos valores serão referências a outras entidades (instâncias de Agent, Resource ou User Defined Type).

Para além do tipo de dados é necessário definir se o atributo é de preenchimento obrigatório (**Required**), se o seu valor é alterável (**ReadOnly**), se o seu valor é persistido (**Persisted**) e se é possível alterar o seu valor após a primeira gravação (**Editable**). É ainda possível definir qual o valor por omissão do atributo, através da propriedade **DefaultValue**.

Existem outras propriedades cujo preenchimento depende do tipo de dados escolhido para o atributo, que serão descritas de seguida. Existe também a possibilidade de descrever fórmulas para definir alguns comportamentos, que serão enumeradas logo após a descrição das propriedades específicas de cada tipo de dados.

Tipos de dados: complexo

É possível, através da propriedade **Cardinality**, descrever a quantidade de instâncias que podem ser associadas a este atributo.

Para além disso, é ainda possível descrever um conjunto de condições (**ComplexTypeConditions**), que são usadas como filtro sobre os dados e que permitem restringir as instâncias do tipo complexo, que são passíveis de serem selecionadas como valor do atributo.

Tipos de dados: número inteiro e número decimal

Pode definir-se através das propriedades **Min** e **Max**, o valor mínimo e máximo, respetivamente, que pode ser atribuído ao atributo.

Tipo de dados: ficheiro

Neste caso, existe a possibilidade de definir se o atributo pode ter vários ficheiros associados ou apenas um (**AllowMultipleFiles**), bem como os tipos de ficheiros (**AllowFileTypes**) que podem ser carregados.

Fórmulas

Através do desenvolvimento de fórmulas, é possível atribuir comportamentos ao atributo, alguns deles relacionados com propriedades já descritas anteriormente.

Com a propriedade **Formula** é possível descrever uma fórmula que permite calcular o valor do atributo. É possível também descrever uma fórmula que valida o valor do atributo (**ValidationFormula**), para além das validações já existentes na linguagem.

É ainda possível desenvolver fórmulas para avaliar se é obrigatório o preenchimento do atributo (**RequiredCondition**)**,** se o valor do mesmo é alterável (**ReadOnlyCondition**) e persistido (**ExistenceCondition**). O resultado destas duas fórmulas sobrepõe o valor definido através das propriedades Required, ReanOnly e Persisted, respetivamente.

4.2.9.2 Lógica de apresentação

Relativamente à organização da interface visual com o utilizador, é possível definir a posição do atributo no ecrã recorrendo a três propriedades: **Row**, **Column**, **Order**. A primeira representa a linha em que o atributo se encontra, a segunda descreve em que coluna e a terceira permite definir uma ordem de apresentação quando o par Row-Column é igual em dois ou mais atributos. Para além da posição, é possível definir o tamanho dos campos nos formulários (**Size**).

A visibilidade pode ser definida através da propriedade **Visible**, sendo que é possível condicionar a visibilidade do atributo por diferentes tamanhos de ecrã. Utilizando a propriedade (**VisibleOnScreenSizes**) é possível definir que um atributo apenas é visível em smartphones, tablets ou computadores.

Existe também a possibilidade de agrupar os atributos por grupos, de forma a facilitar a organização da interface visual. Para isso, é necessário definir a propriedade **Group**. Os atributos com um valor igual nesta propriedade são considerados do mesmo grupo.

Caso o atributo se encontre numa grelha (caso faça parte do tipo de compromisso/evento que corresponde à propriedade Details de um Interaction Type), existem duas propriedades adicionais que podem ser definidas: **IsDetail** e **InDetailsSize**. A primeira representa que o atributo será colocado numa secção de detalhes que não está sempre visível; enquanto a segunda descreve o tamanho do campo nessa mesma secção de detalhes.

Fórmulas

De forma a condicionar a visibilidade do atributo é possível descrever uma fórmula para esse efeito (**VisibleCondition**) e que sobrepõe o valor da propriedade Visible.

4.2.9.3 Comportamento

Com a estrutura representada anteriormente é possível descrever Attributes com diferentes configurações. De forma a tornar o modelo mais completo, é possível atribuir comportamentos aos Attributes. Desta forma, surgem os conceitos Trigger e Action.

Para cada Attribute é possível definir múltiplos Triggers (relação 1-N), cada um deles também com múltiplas Actions (relação 1-N).

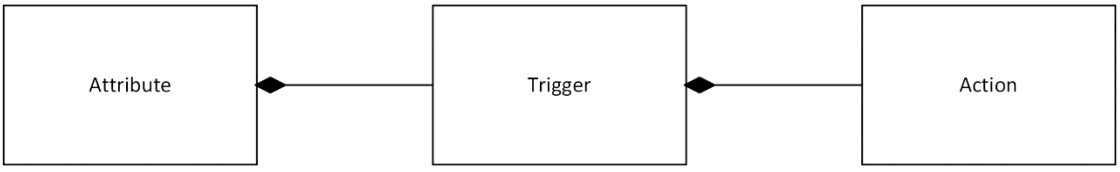


Figura 23 - Relação Attribute - Trigger - Action

O conceito Trigger descreve um momento do tempo de vida do Attribute em que algo acontece e engloba em si um conjunto de ações (Actions) a executar. Assim sendo, existem dois tipos de Triggers diferentes: Create (momento de criação do Attribute) e Change (momento em que o valor de Attribute é alterado).

A sua descrição é feita tendo em conta apenas a propriedade **Type** (como referido anteriormente, apenas tem dois valores possíveis: Create e Change).

Em relação a cada Action, esta pode ter tipos diferentes (**Type**), de acordo com o comportamento que se pretende atribuir. Assim, os tipos existentes são os seguintes:

- *Get user agent*: Permite obter informações sobre o utilizador atual, quando este se encontra relacionado com uma instância de um Agent Type;
- *Get number*: No caso de o Attribute se encontrar num Interaction Type, permite obter o próximo número identificador disponível;
- *Get entity*: Permite obter informações de uma entidade;
- *Get entity item*: Permite obter informações de um Entity Item;
- *Get entity message*: Permite obter um texto que se encontra num atributo de uma entidade e mostrá-lo como mensagem de ajuda ao utilizador;
- *Get accounts*: Permite obter informações de contas relacionadas com a contabilidade;
- *Get transactional entities*: No caso de o Attribute se encontrar num Interaction Type, permite obter Commitment/Event Types e adicionar esses registos à grelha presente no ecrã;
- *Formula*: Descreve uma fórmula para ser avaliada;
- *Check*: No caso de o Attribute ser do tipo booleano, quando o valor é alterado, permite atribuir a outros Attributes novos valores.

Para além do tipo de Action, é importante definir as propriedades que permitem identificar o tipo de entidade, necessárias para obter os dados corretos. Essas propriedades são **Kind** e **Entity Type**. Os valores possíveis destas duas propriedades dependem do tipo da Action em questão. Assim sendo, os valores possíveis para a propriedade Kind para cada um dos diferentes Types são os seguintes, demonstrados pela tabela.

Tabela 1 - Valores possíveis da propriedade Kind por cada Type

Type	Agent	Resource	User Defined	Commitment	Event
Get entity	X	X	X		
Get entity item	X	X	X		
Get entity message	X	X	X		
Get accounts	X	X			
Get transactional entities				X	X

É ainda possível definir uma lista de correspondências entre os **Attributes** que se pretendem obter dos serviços da plataforma e aqueles que se encontram na entidade atual. Adicionalmente, é possível definir um conjunto de condições (**Conditions**) para serem utilizadas como filtro para selecionar os dados corretamente.

4.2.10 Sistema de contabilidade

Visto que a BAMoL assenta sobre uma framework de contabilidade (REA), é possível descrever na linguagem mecanismos que permitem ter na aplicação a movimentação de contas contábeis.

Assim sendo, existem três coisas a descrever: os Account Types, os Accounting Triggers e as suas Accounting Actions. Enquanto os primeiros referem a definição das contas, os segundos descrevem a forma como as mesmas são movimentadas. Os Accounting Triggers são sempre associados aos Types que descrevem transações, ou seja, Commitment e Event Types. Todos estes conceitos são descritos em seguida e as suas relações representadas na próxima figura.

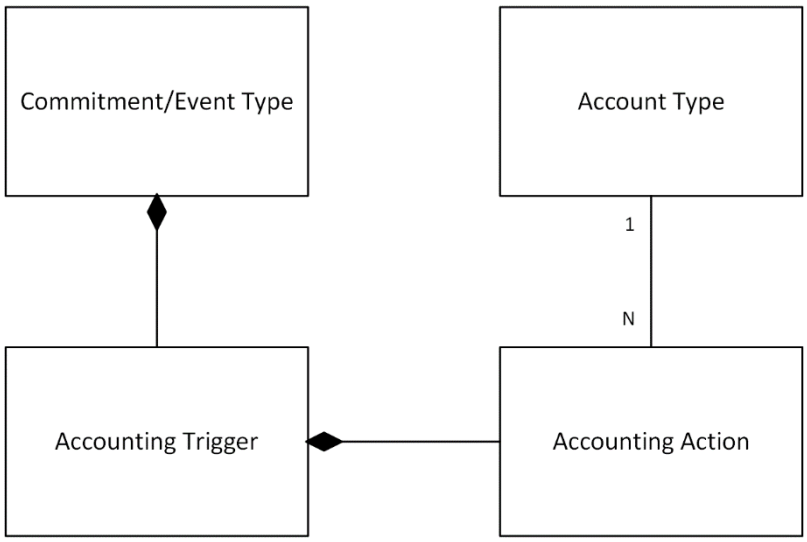


Figura 24 - Relação entre conceitos de Accounting

4.2.10.1 Account Type

A definição de cada Account Type é feita através de apenas três propriedades: **ApplyTo**, **Category** e **ExportToExternalAccount**.

A primeira representa que tipo de conta se pretende descrever e tem quatro valores possíveis:

- Resource: Representa os recursos controlados pela empresa;
- Agent: Representa os compromissos entre o agente económico e a empresa;
- Revenue: Descreve uma conta que representa um proveito;
- Expense: Descreve uma conta que representa um custo.

No caso da propriedade Category, esta escreve a forma como se irá efetuar a ligação à contabilidade geral e contém cinco valores possíveis: Assets, Liabilities, Revenues, Expenses e Other.

A última das três propriedades indica se existe uma conta correspondente no plano de contas do ERP.

4.2.10.2 Accounting Triggers

Como foi dito anteriormente, este conceito permite descrever, juntamente com o conceito que é apresentado logo de seguida, o conjunto de operações que irão ser realizadas na movimentação das contas, num determinado momento. Esse momento pode ser definido como sendo sempre que existe uma transação, ou seja, os Accounting Triggers serão associados aos Commitment ou Event Types. Desta forma, pode-se resumir o funcionamento deste conceito da seguinte forma: sempre que existe a transação de um Resource Type, existe um conjunto de ações que serão executadas, de forma a movimentar as diversas contas que estão configuradas para o efeito.

Para tal, é necessário descrever, para além de uma lista de Accounting Actions (tantos quantas contas se pretendem movimentar), duas fórmulas que ao serem avaliadas permitem obter a empresa para a qual será realizado o movimento de contas (**CompanyRule**) e o montante a movimentar (**ValueRule**). Para além dessas propriedades, é necessário definir o momento em que a operação será executada (na criação ou na aprovação), através de **OnOperation**.

É também possível condicionar o processamento do Accounting Trigger através de uma fórmula definida em **OnCondition**.

4.2.10.3 Accounting Actions

Cada Accounting Action descreve a movimentação de uma conta, num determinado lote de operações, representado por cada Accounting Trigger.

Para a descrição deste elemento é necessário definir qual o **Account Type** que será movimentado e qual o tipo de movimento a realizar (**MovementKind**)**:** adição ou subtração. Para além disso, caso represente um reconhecimento de proveitos, é necessário atribuir um de dois valores possíveis (Expense ou Revenue) à propriedade **RecognitionKind**.

Por fim, é necessário descrever os sufixos das contas a utilizar, tanto na plataforma myMIS como no ERP, caso seja utilizado. Para o primeiro caso, através de **AccountNameSuffix** é possível descrever uma fórmula, que ao ser avaliada permite conhecer o sufixo da conta na plataforma myMIS a ser movimentada. No caso do ERP, o sufixo da conta é descrito na propriedade **ExternalAccountNameSuffix**.

4.2.11 Mecanismo de aprovações

Na BAMoL é possível definir diferentes etapas de aprovação para entidades de negócio que necessitem deste mecanismo. Para tal, é necessário definir as etapas de aprovação (designadas por Approval Stages) para cada um dos Agent, Resource, User Defined e Interaction Types.

Com o conceito Approval Stage, é possível definir um fluxo complexo de aprovações com várias etapas, que permite a quem desenvolve os modelos uma grande liberdade na descrição dos mecanismos de aprovação das suas entidades de negócio.

Cada um desses fluxos deve conter pelo menos dois Approval Stages, em que um deles representa a etapa inicial (**IsStart**) e o outro a etapa final (**IsEnd**).

Em cada etapa é necessário definir qual a etapa seguinte em caso de aprovação (**WhenApproved**) e em caso de rejeição (**WhenRejected**). É também possível definir uma fórmula (**EvaluateExecution**) que permite avaliar se a etapa pode ser descartada no circuito de aprovação. Caso não seja definida, a etapa será sempre incluída.

Para além disso, é necessário definir um conjunto de regras (**ApproverRules**), através de fórmulas, que permitem calcular quem será o próximo aprovador a intervir no circuito de aprovação.

Caso seja necessário a alteração do valor de alguns atributos na etapa de aprovação, é possível defini-los através da propriedade **EditableAttributes**.

4.2.11.1 Policy Type

Este conceito surge por forma a complementar o anterior. Isto é, ao descrever etapas de aprovação, aparece consequentemente a necessidade de definir regras que realizem essa mesma aprovação de forma automática. Com este conceito é possível definir tipos de regras de aprovação. Cada Policy Type é descrito no contexto de um Event ou Commitment Type, pertencente a um Interaction Type e a um Process Type.

Para o desenvolvimento deste conceito, é necessário descrever uma lista (**Attributes**) em que para cada elemento da mesma é definido o atributo do Commitment/Event Type e um operador de comparação. Cada um destes elementos irá ser utilizado para definir as condições em que as regras são utilizadas.

É também necessário definir três propriedades que contêm informação sobre a conta da contabilidade que serve de controlo para que a regra se aplique ou não. Essas propriedades são **AccountName** (que permite definir uma fórmula para conhecer qual o prefixo da conta a verificar), **Operator** (operador a utilizar na comparação) e **Period** (período a avaliar, que pode ter um dos seguintes valores: dia, mês ou ano).

É possível também definir um conjunto de centros de responsabilidade (**ResponsibilityCenters**), cujo código servirá de prefixo para compor o nome da conta que será utilizada para efetuar a comparação dos valores.

4.3 Conceitos complementares

4.3.1 Mecanismo de notificações

De forma a ser possível notificar os utilizadores do sistema de que algo aconteceu, seja por alguma ação realizada por si, por um outro utilizador ou pelo sistema, surge o conceito da Notification.

Atualmente, uma Notification é uma mensagem de email que é enviada a um utilizador quando ocorre uma determinada ação. É possível definir o envio dessas mensagens para ações realizadas sobre Agent, Resource, User Defined e Interaction Types.

As propriedades utilizadas para definir uma notificação podem dividir-se em dois grupos. O primeiro descreve o momento em que a mensagem é enviada e o segundo permite descrever os destinatários e o conteúdo da mensagem a enviar.

Assim, para definir quando a notificação é enviada, é necessário descrever o momento, através da propriedade **On** (que tem cinco valores possíveis: Create, Update, Go to approve, Revert e Automatic approve), qual o **ApprovalStage** (no caso de o valor da propriedade On ser 'Go to approve', 'Revert' ou 'Automatic approve') e uma fórmula que representa uma condição que permite saber se a notificação deve ser enviada ou não (**FiredOnCondition**). Esta última propriedade é opcional sendo que, caso não seja preenchida, as notificações serão sempre enviadas.

Os endereços dos destinatários (**To**) da notificação são descritos através de uma fórmula (que permite obter o endereço de destino da mensagem) ou de endereços de email fixos. A mesma lógica é utilizada para mais duas propriedades: **CC** e **BCC**.

Em relação ao conteúdo da mensagem, o mesmo é dividido em duas partes: o assunto (**Subject**) e o corpo da mensagem (**Body**). Em ambos os casos é possível definir vários parâmetros, que permitem que o conteúdo das mensagens não seja fixo. Para tal, é necessário definir as propriedades **SubjectParameters** e **BodyParameters**, que contém um conjunto de atributos, cujo valor será utilizado para compor o conteúdo de Subject e Body, respetivamente. Para que essa utilização aconteça, as propriedades Subject e Body devem conter o texto "{N}" (sem aspas) no local em que se pretende que o valor dos parâmetros seja colocado, em que N representa a posição do parâmetro na lista, começando no número 0.

4.3.2 Listagens

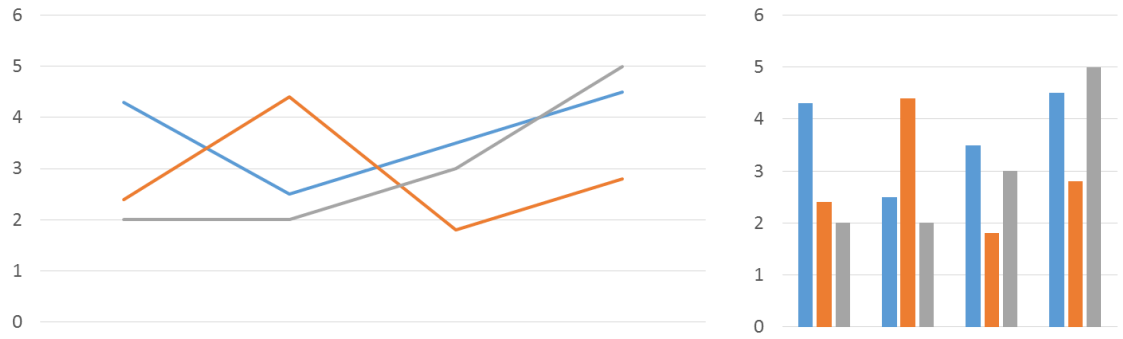
Com este conceito é possível descrever, para um determinado tipo de entidade, uma forma de ver informação sobre o mesmo no formato de listagem, sendo exequível descrever vários elementos do tipo List, que irão representar formas diferentes de ver os detalhes das instâncias do tipo de entidade em questão.

Para tal, é necessário descrever a condição (**Condition**) utilizada para filtrar os dados e uma lista de campos (**Fields**) que representam as colunas das listagens.

Para cada um desses campos é necessário definir qual o atributo a que se refere. Para além disso, é possível descrever qual o tipo de ordenação (**SortType**), que pode ser ascendente ou descendente, qual a prioridade deste campo no momento da ordenação (**SortOrder**), se é um campo de agrupamento de informação (**GroupBy**) e em que tamanhos de ecrã o campo estará visível (**VisibleOnScreenSizes**).

4.3.3 Painéis de controlo

Um painel de controlo (dashboard) permite efetuar um conjunto de consultas para os dados de um determinado Event ou Commitment Type, que se encontra num determinado Interaction Type e num Process Type. A informação é visualizada através de uma tabela e de dois gráficos, cujo esquema de visualização pode ser visto na figura seguinte.



Coluna 1	Coluna 2	Coluna 3	Coluna 4	Coluna 5
Valor 1	Valor 2	12/03/2014	80	13
Valor 1	Valor 3	14/03/2014	84	34
Valor 1	Valor 4	25/03/2014	49	21

Figura 25 - Painel de controlo: esquema de representação

Para a descrição deste conceito são utilizadas as propriedades **Condition** (condição para filtrar os dados), **NumberOfRecords** (número de registos que se pretendem obter), **CompanyCodeField** (atributo do Commitment/Event Type que permite saber qual a empresa) e **Dimensions** (conjunto de dimensões pelas quais é possível agrupar os dados).

Para a tabela é necessário descrever cada coluna que nela estará presente. Para cada uma dessas colunas é necessário definir qual o atributo a que se refere. Para além disso, é possível descrever qual o tipo de ordenação (**SortType**), que pode ser ascendente ou descendente, qual a prioridade dessa coluna no momento da ordenação (**SortOrder**), se é uma coluna de agrupamento de informação (**GroupBy**) e qual o tipo de agregação de dados (**AggregateType**) (soma, média ou contagem).

Em relação a cada um dos gráficos, é necessário descrever qual o tipo de gráfico (**Type**), que pode ser de barras ou linhas. Além disso, é necessário indicar qual o atributo do Event/Commitment Type que representa as categorias do gráfico (eixo dos XX) (**InitialCategory**) e qual o atributo que representa os valores das séries (eixo dos YY) (**Series**).

4.3.4 Ferramentas

Existem várias ferramentas que podem ser adicionadas ao modelo de forma a acrescentar funcionalidades adicionais às aplicações e isso é descrito utilizando o conceito Tool.

As ferramentas que podem ser atualmente escolhidas são descritas de seguida:

- *Ficheiros*: Permite efetuar uploads de ficheiros, bem como ter uma área por utilizador em que são guardados os ficheiros carregados;
- *Geração de SAFT-PT*: Geração e exportação do ficheiro SAFT-PT[5];
- *Geração de ficheiro de contabilidade*: Geração e exportação do ficheiro de contabilidade que permite uma posterior integração de dados num ERP;
- *Diagnóstico de diferenças*: Permite o diagnóstico entre o sistema myMIS e um sistema externo (ERP). Deteta as diferenças relativas a Resource, Agent e User Defined Types.

4.3.5 Configurações adicionais

Através da descrição de configurações adicionais (em XML) é possível na BAMoL efetuar algumas ações que, através desta forma, são alteráveis mais facilmente, de forma a ir de encontro às necessidades do problema. É possível descrever configurações para as mais variadas situações, desde a forma como é feita a integração com um ERP externo, até descrever como vão ser impressas as instâncias de um Interaction Type.

4.4 Modelo resultante do cenário de exemplo

Partindo do problema apresentado no cenário de exemplo e da descrição da linguagem realizada anteriormente, é apresentado neste capítulo o modelo resultante.

A notação utilizada permite, posteriormente, desenvolver o modelo na ferramenta de desenvolvimento de uma forma fácil, pois a linguagem utilizada tanto na ferramenta como nos esquemas seguintes é bastante aproximada.

[5] Standard Audit File for Tax purposes (Versão Portuguesa) - http://info.portaldasfinancas.gov.pt/pt/apoio_contribuinte/NEWS_SAF-T_PT.htm

Agent Types

Figura 26 - Modelo resultante: Agent Types

Resource Types

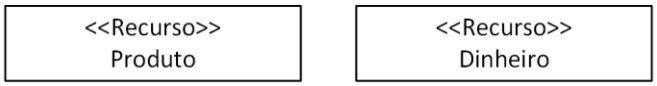


Figura 27 - Modelo resultante: Resource Types

User Defined Types

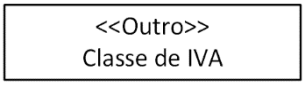


Figura 28 - Modelo resultante: User Defined Types

Commitment e Event Types

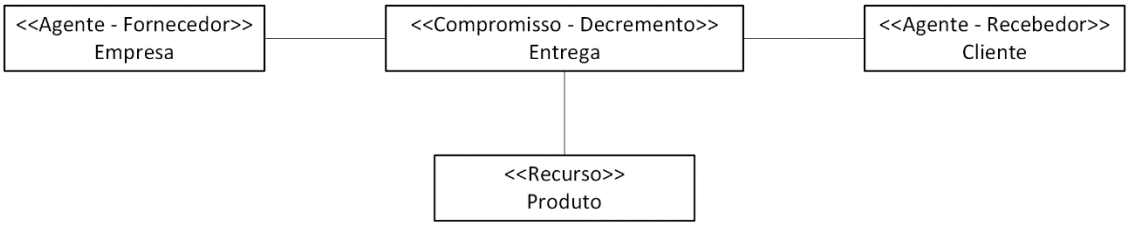


Figura 29 - Modelo resultante: Commitment Type – Entrega

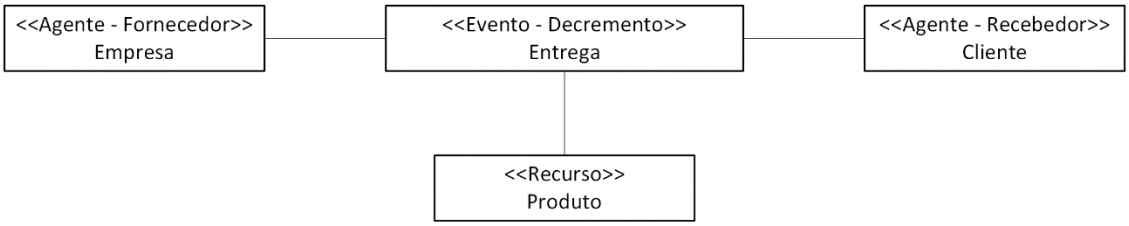


Figura 30 - Modelo resultante: Event Type – Entrega

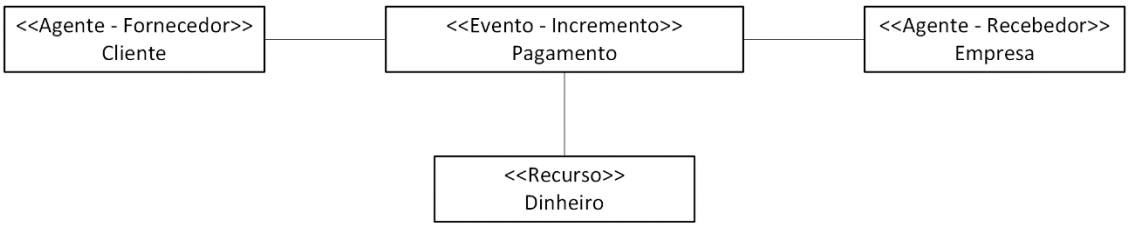


Figura 31 - Modelo resultante: Event Type – Pagamento

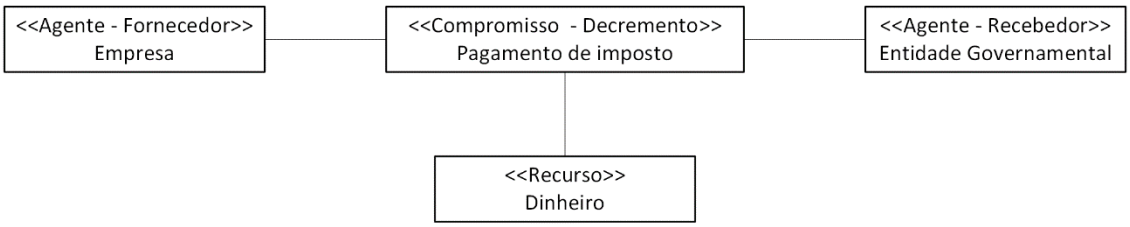


Figura 32 - Modelo resultante: Commitment Type – Pagamento de imposto

Interaction Types

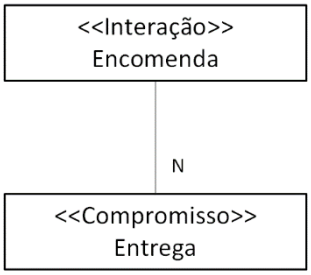


Figura 33 - Modelo resultante: Interaction Type – Encomenda

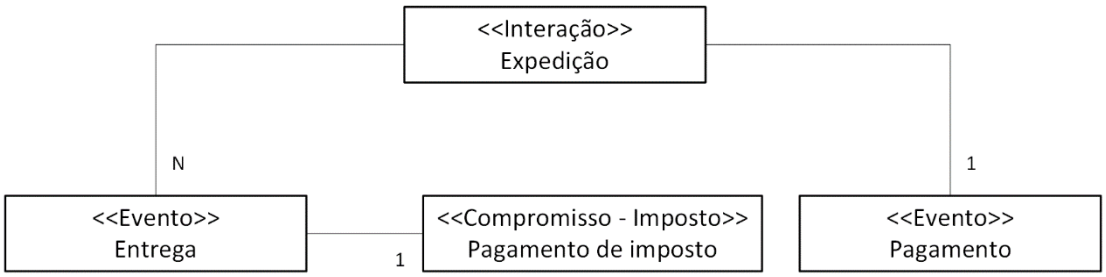


Figura 34 - Modelo resultante: Interaction Type – Expedição

Process Types

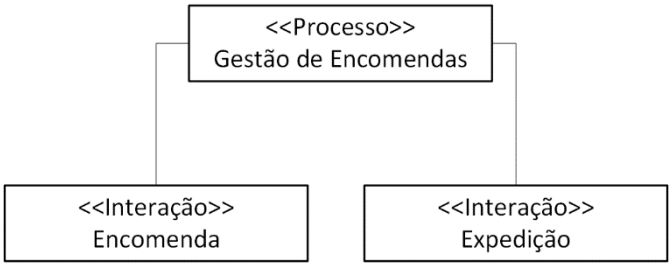


Figura 35 - Modelo resultante: Process Types

4.5 Gramática da linguagem

A partir da descrição textual da BAMoL foi definida uma gramática da linguagem, indispensável para a formalização da mesma. Essa gramática será utilizada para realizar a validação sintática dos modelos, que será explicada mais à frente neste mesmo documento.

A gramática foi criada num formato XML Schema (XSD), pois pretende-se que futuramente os modelos presentes na plataforma myMIS sejam persistidos num formato XML. Desta forma, torna-se mais simples o processo de validação, visto que as tecnologias utilizadas para a persistência de modelos e para a sua validação são totalmente compatíveis.

De seguida é apresentado um excerto da gramática, bem como o modelo correspondente. Estes trechos de código correspondem à validação e modelo de um Agent Type, de acordo com a descrição feita anteriormente neste capítulo.

```
<xs:complexType name="EntityType">
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Attributes">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Attribute" maxOccurs="unbounded"
minOccurs="0" type="mymis:Attribute"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="EntityItems">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="EntityItem" maxOccurs="unbounded"
minOccurs="0" type="mymis:EntityItemType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="ApprovalStages">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Stage" maxOccurs="unbounded" minOccurs="0"
type="mymis:ApprovalStage"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="IsResponsibilityCenterType" type="xs:boolean"/>
      <xs:element name="IsCompanyType" minOccurs="0" maxOccurs="1"
type="xs:boolean"/>
    </xs:sequence>
    <xs:attribute name="Code" type="xs:ID" use="required" />
  </xs:complexType>
```

Código 1 - Excerto da gramática da linguagem

```
<EntityType Code="myCompany">
      <Name>myCompany</Name>
```

```
<Attributes>
  <Attribute>
    <Code>myCompany_Phone</Code>
    <Name>Phone</Name>
    <Description />
    <Type>ST1</Type>
    <Required>false</Required>
    <RequiredCondition>
      <Elements />
    </RequiredCondition>
    <ExistenceCondition>
      <Elements />
    </ExistenceCondition>
    <Min>0</Min>
    <Max xsi:nil="true" />
    <DefaultValue />
    <Editable>true</Editable>
    <Formula>
      <Elements />
    </Formula>
    <ValidationFormula>
      <Elements />
    </ValidationFormula>
    <AllowMultipleFiles>false</AllowMultipleFiles>
    <AllowFileTypes />
    <ReadOnly>false</ReadOnly>
    <ReadOnlyCondition>
      <Elements />
    </ReadOnlyCondition>
    <Visible>true</Visible>
    <VisibleCondition>
      <Elements />
    </VisibleCondition>
    <Triggers />
  </Attribute>
  <Attribute>
    <Code>myCompany_Fax</Code>
    <Name>Fax</Name>
    <Description />
    <Type>ST1</Type>
    <Required>false</Required>
    <RequiredCondition>
      <Elements />
    </RequiredCondition>
    <ExistenceCondition>
      <Elements />
    </ExistenceCondition>
    <Min>0</Min>
    <Max xsi:nil="true" />
    <DefaultValue />
    <Editable>true</Editable>
    <Formula>
      <Elements />
    </Formula>
    <ValidationFormula>
      <Elements />
```

```
    </ValidationFormula>
    <AllowMultipleFiles>false</AllowMultipleFiles>
    <AllowFileTypes />
    <ReadOnly>false</ReadOnly>
    <ReadOnlyCondition>
      <Elements />
    </ReadOnlyCondition>
    <Visible>true</Visible>
    <VisibleCondition>
      <Elements />
    </VisibleCondition>
    <Triggers />
  </Attribute>
  <Attribute>
    <Code>myCompany_Code</Code>
    <Name>Code</Name>
    <Description />
    <Type>ST1</Type>
    <Required>true</Required>
    <RequiredCondition>
      <Elements />
    </RequiredCondition>
    <ExistenceCondition>
      <Elements />
    </ExistenceCondition>
    <Min>0</Min>
    <Max xsi:nil="true" />
    <DefaultValue />
    <Editable>true</Editable>
    <Formula>
      <Elements />
    </Formula>
    <ValidationFormula>
      <Elements />
    </ValidationFormula>
    <AllowMultipleFiles>false</AllowMultipleFiles>
    <AllowFileTypes />
    <ReadOnly>false</ReadOnly>
    <ReadOnlyCondition>
      <Elements />
    </ReadOnlyCondition>
    <Visible>true</Visible>
    <VisibleCondition>
      <Elements />
    </VisibleCondition>
    <Triggers />
  </Attribute>
  <Attribute>
    <Code>myCompany_Name</Code>
    <Name>Name</Name>
    <Description />
    <Type>ST1</Type>
    <Required>true</Required>
    <RequiredCondition>
      <Elements />
    </RequiredCondition>
    <ExistenceCondition>
```

```
            <Elements />
          </ExistenceCondition>
          <Min>0</Min>
          <Max xsi:nil="true" />
          <DefaultValue />
          <Editable>true</Editable>
          <Formula>
            <Elements />
          </Formula>
          <ValidationFormula>
            <Elements />
          </ValidationFormula>
          <AllowMultipleFiles>false</AllowMultipleFiles>
          <AllowFileTypes />
          <ReadOnly>false</ReadOnly>
          <ReadOnlyCondition>
            <Elements />
          </ReadOnlyCondition>
          <Visible>true</Visible>
          <VisibleCondition>
            <Elements />
          </VisibleCondition>
          <Triggers />
        </Attribute>
      </Attributes>
      <EntityItems />
      <ApprovalStages />
      <IsResponsibilityCenterType>false</IsResponsibilityCenterType>
      <IsCompanyType>true</IsCompanyType>
    </EntityType>
```

Código 2 - Excerto de modelo

A gramática completa é passível de consulta na secção de anexos deste documento (Anexo 1 - Gramática da linguagem BAMoL no formato XML Schema).

5 Validação sintática de modelos

A falta de mecanismos de validação de modelos é um dos problemas atuais no desenvolvimento de aplicações na plataforma, pois sem isso a probabilidade de cometer erros na criação ou alteração dos modelos aumenta. Para tal e de forma a diminuir a possibilidade de ocorrência de erros, existe a necessidade de validar os modelos sempre que aconteçam novos desenvolvimentos nos mesmos.

Para que um modelo seja interpretado pela plataforma myMIS de forma correta, é necessário que este seja criado sob certas regras. Estas são definidas pela gramática da linguagem (representada no formato XSD) e cada modelo tem que ser válido perante essas mesmas normas.

Para além das validações que a gramática permite realizar, são necessárias outras adicionais, que são impossíveis de efetuar recorrendo à tecnologia do XML Schema. Assim, de forma a ser possível validar totalmente o modelo, é necessário recorrer a outra tecnologia que permite fazer as restantes validações, sendo que para isso são utilizadas as *Extensible Stylesheet Language Transformations* (XSLT).

Na imagem seguinte é esquematizado todo o processo de validação de modelos, de forma a ser mais percetível o funcionamento do mesmo.

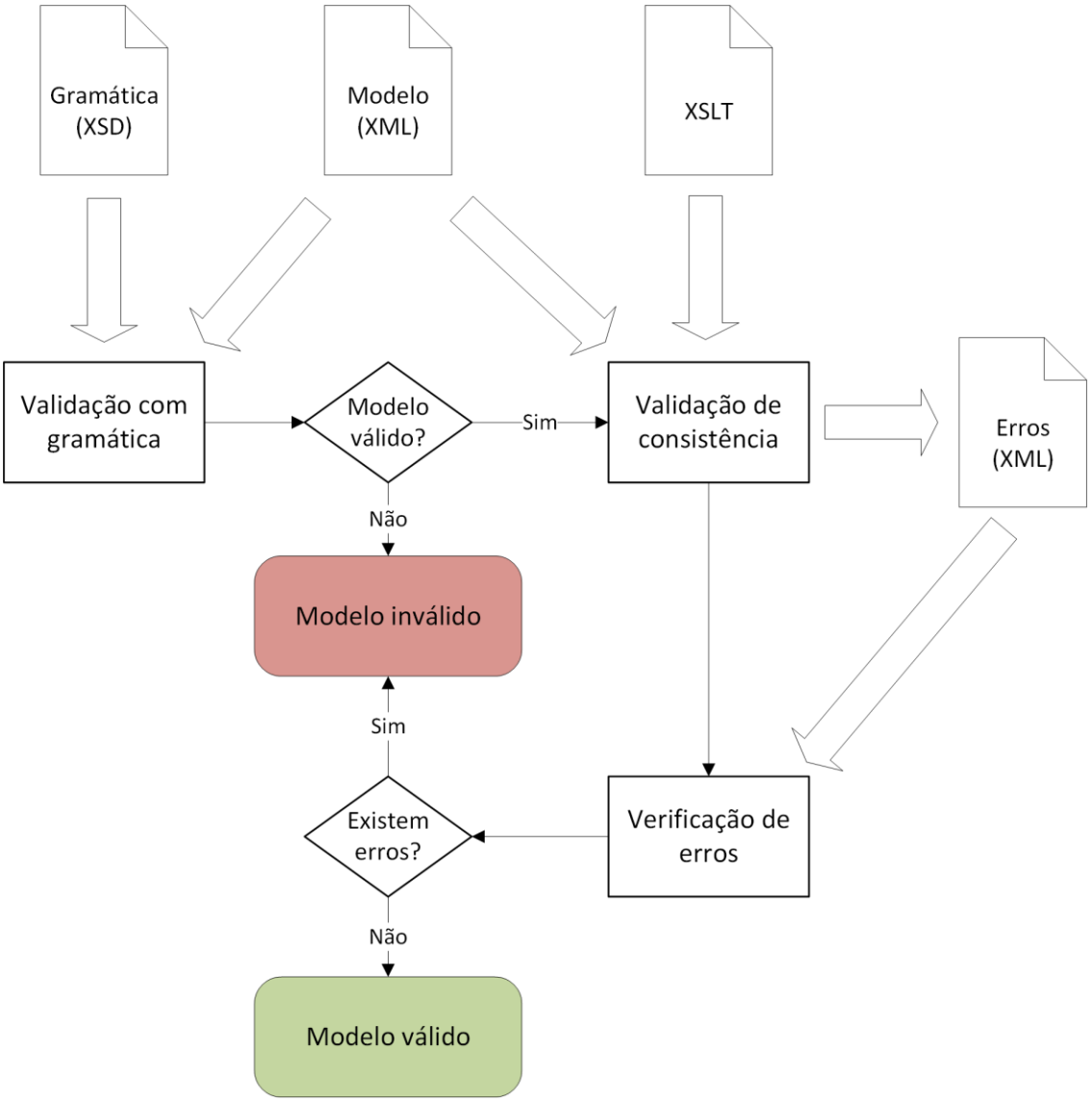


Figura 36 - Processo de validação sintática de modelos

Com o auxílio da tabela seguinte podemos perceber quais os inputs e outputs de cada um destes passos presentes na validação de um modelo.

Tabela 2 - Processo de validação sintática de modelos (inputs e outputs)

	Inputs	**Outputs**
Validação com gramática	Gramática (XSD) + Modelo (XML)	
Validação de consistência	Modelo (XML) + XSLT	Erros (XML)
Verificação de erros	Erros (XML)	

Utilizando a imagem e a tabela anteriores, podemos resumir o funcionamento deste mecanismo de validação nos seguintes passos:

1. Validação com gramática: O modelo em formato XML é confrontado com a gramática e neste ponto é possível encontrar vários erros a nível sintático, como são o caso de propriedades em falta ou a referência a elementos não existentes ou inválidos;

2. Validação de consistência: O modelo após passar a primeira validação é submetido a uma transformação utilizando um ficheiro XSLT, em que irá resultar um ficheiro XML que contém todos os erros encontrados nesta fase. Os erros passíveis de serem detetados neste ponto são declarados mais à frente neste documento.

5.1 Validação de consistência

Após a primeira validação, utilizando a gramática da linguagem, é necessário perceber se o modelo, para além de estar descrito com a estrutura correta, contém um conteúdo coerente. E isso é feito na segunda validação do modelo. Em seguida são descritas as validações realizadas neste ponto.

Acessibilidade dos atributos

Estas validações têm como objetivo saber se existem atributos em que o seu preenchimento é impossível de realizar. Neste caso existem dois cenários a considerar:

1. Atributo não visível, sem qualquer forma de se obter o seu valor (seja através de uma fórmula ou de uma ação de um Trigger de um outro atributo);

2. Atributo cujo valor é inalterável e também não existe forma de o calcular (como no caso anterior, através de uma fórmula ou de uma ação de um Trigger de um outro atributo).

Em ambos os casos, o atributo em questão está configurado de tal maneira que é impossível ao mesmo ter qualquer valor atribuído. Desta forma, existe um erro de desenvolvimento, em que o atributo não tem qualquer utilidade.

Fórmulas

De forma a serem interpretadas corretamente, é necessário que as fórmulas desenvolvidas no modelo se encontrem escritas corretamente. Para tal, é necessário que estas cumpram alguns requisitos, nomeadamente:

1. Em caso de funções, que as mesmas façam parte do conjunto das funções suportadas pela plataforma, que o número de parâmetros seja correto e que o seu tipo de dados seja correto para a função em causa;

2. Os operadores sejam válidos;

3. Os atributos utilizados façam parte do contexto em que os mesmos se encontram e que os tipos de dados sejam compatíveis.

Incompatibilidade de compromissos/eventos

Num Interaction Type podem existir dois tipos de compromissos ou eventos associados que o constituem. De forma que o modelo seja válido, esses dois elementos (Details e Summary) nunca podem estar relacionados com o mesmo tipo de compromisso/evento. Para além disso, caso esses dois elementos tenham a mesma natureza (sejam ambos tipo de compromisso ou ambos tipo de evento), a propriedade Kind do tipo de compromisso/evento nunca pode ter o mesmo valor.

5.2 Ferramenta de validação

De forma a ser possível realizar as validações necessárias aos modelos, foi criada uma ferramenta de apoio à equipa de desenvolvimento e que segue exatamente o fluxo apresentado anteriormente. Esta ferramenta permite saber se os modelos desenvolvidos são válidos e se, por outro lado, com as atualizações feitas ao longo do tempo estes têm informação que não seja necessária e que, por algum motivo, não foi eliminada do modelo.

O fluxo de processamento que esta ferramenta segue é o representado na figura seguinte.

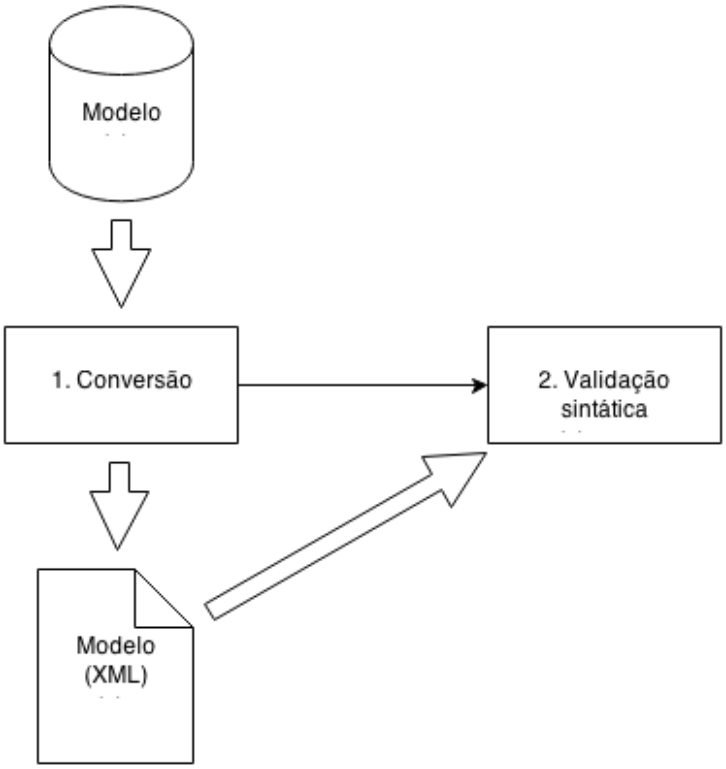


Figura 37 - Ferramenta de validação de modelos: fluxo de processamento

Na figura anterior temos em 1 a representação do processo de conversão do modelo que se encontra na estrutura de persistência da plataforma, para a sua representação correspondente no formato XML. Na ação representada por 2 ocorre a validação sintática dos modelos, como explicado no início deste capítulo.

5.2.1 Obtenção do modelo em formato XML

A realização das validações sintáticas de modelos necessitam que os mesmos se encontrem no formato XML. Para tal, é necessário converter os modelos persistidos na base de dados da plataforma para esse mesmo formato.

Essa conversão é realizada mapeando a estrutura de persistência utilizada pela plataforma para um ficheiro XML que cumpre as regras estruturais definidas pela gramática da linguagem.

O mapeamento de cada uma das propriedades dos elementos do modelo é, em quase todos os casos, realizado de uma forma direta, isto é, os tipos de dados na estrutura de persistência da plataforma são os mesmos na representação do modelo no formato XML. As únicas exceções são as propriedades que são representativas de fórmulas.

Mapeamento de fórmulas

As fórmulas são guardadas na estrutura de persistência da plataforma como texto simples. Através dessa representação é extremamente difícil, através do formato XML, validar cada uma das fórmulas. Assim sendo, cada uma das propriedades que representam fórmulas no modelo foram convertidas numa estrutura que permite facilitar a validação através de XML Schema (XSD) e de XSLT.

Essa conversão utiliza um algoritmo criado para o efeito e que é apresentado de seguida.

```
MÉTODO ObterElementosDeFórmula
  INPUT: formula (Texto)
  VARIÁVEL: elementos (Lista)
  SE (é_Função(formula))
    Função func = obterTipoDeFunção(formula)
    func.Parametros = obterParametrosDeFuncao(formula)
    elementos.Adiciona(func)
  SENÃO
    Lista elementos = converterEmListaDeElementos(formula)
    INTEIRO n = 0
    ENQUANTO(n < elementos.NúmeroDeElementos)
         TEXTO el = elementos[n]
         SE (é_Operador(el))
             Operador op = criarOperador(el)
             elementos.adiciona(op)
         SENÃO
             SE (é_ElementoComposto(el))
                  INTEIRO m, TEXTO el_aux = el
                  PARA (m = n + 1 ATÉ elementos.NúmeroDeElementos)
                           el_aux = el_aux + elementos[m]
                  FIM PARA
                  Lista elems = ObterElementosDeFórmula(el_aux)
                  ElementoComposto ec = criarElementoComposto(elems)
```

```
                elementos.adiciona(ec)
                n = m
            SENÃO
                Elemento = criarElementoSimples(el)
                elementos.adiciona(el)
            FIM SE
        FIM SE
        n = n + 1
    FIM ENQUANTO
  FIM SE
  DEVOLVE elementos
FIM MÉTODO
```

Código 3 - Algoritmo de conversão de fórmulas

Em seguida, é possível observar exemplos do resultado do mapeamento de fórmulas, obtidos através do algoritmo anterior, cujos valores originais são 'SUM([Details.Amount])', '[Amount]-[Incidence]' e '[Amount]/(1+([VATPercentage]*0.01))'.

```
<Formula>
  <Elements>
    <Function>
      <Code>SUM</Code>
      <Parameters>
        <Element>
          <Location>Details</Location>
          <Value>Amount</Value>
          <Elements />
        </Element>
      </Parameters>
    </Function>
  </Elements>
</Formula>
```

Código 4 – Mapeamento de fórmulas: exemplo 1 - SUM([Details.Amount])

```
<Formula>
  <Elements>
    <Element>
      <Location>Self</Location>
      <Value>Amount</Value>
      <Elements />
    </Element>
    <Operator>-</Operator>
    <Element>
      <Location>Self</Location>
      <Value>Incidence</Value>
      <Elements />
    </Element>
  </Elements>
</Formula>
```

Código 5 – Mapeamento de fórmulas: exemplo 2 - [Amount]-[Incidence]

```
<Formula>
  <Elements>
    <Element>
      <Location>Self</Location>
      <Value>Amount</Value>
      <Elements />
    </Element>
    <Operator>/</Operator>
    <Element>
      <Location />
      <Elements>
        <Element>
          <Location />
          <Value>1</Value>
          <Elements />
        </Element>
        <Operator>+</Operator>
        <Element>
          <Location />
          <Elements>
            <Element>
              <Location>Self</Location>
              <Value>VATPercentage</Value>
              <Elements />
            </Element>
            <Operator>*</Operator>
            <Element>
              <Location />
              <Value>0.01</Value>
              <Elements />
            </Element>
          </Elements>
        </Element>
      </Elements>
    </Element>
  </Elements>
</Formula>
```

Código 6 - Mapeamento de fórmulas: exemplo 3 - [Amount]/(1+([VATPercentage]*0.01))

5.2.2 Atividade experimental: Validação de modelo existente em produção

Com o objetivo de verificar a qualidade deste processo, foi validado um modelo atualmente em produção na plataforma e com vários utilizadores, utilizando esta técnica. Desta forma, para além de validar a ferramenta de validação de modelos, foi também possível perceber o estado em que se encontrava o modelo em causa.

Assim, utilizando o processo e técnicas descritas anteriormente, o modelo foi validado, sendo possível perceber a existência de alguns atributos que não são utilizados em momento algum da utilização da aplicação, como é possível ver no excerto seguinte do resultado da validação.

```
<Error>Expense Report - Attribute not reachable: ApprovalStatus</Error>
<Error>Expense Report - Attribute not reachable: ApprovalNote</Error>
```

Código 7 – Excerto de resultado de validação

O modelo utilizado, na primeira validação não obteve nenhum erro, sendo estes encontrados aquando da segunda validação.

Após a validação utilizando a ferramenta, foi verificado, utilizando a base de dados da plataforma e a interface com o utilizador, se os erros encontrados pela ferramenta desenvolvida são fundamentados. Pôde verificar-se que sim, ou seja, a ferramenta encontrou atributos que realmente não se encontravam acessíveis nem eram utilizados na aplicação em análise.

Através de uma análise posterior foi possível perceber que alguns dos erros identificados pela aplicação de validação relacionados com a acessibilidade dos atributos, foram originados pela evolução da plataforma, em que esses atributos deixaram de ser necessários e não foram removidos.

5.2.2.1 Conclusões da atividade

Esta atividade experimental permitiu retirar duas conclusões: a aplicação criada para validar os modelos permite realmente encontrar falhas nos modelos; o modelo em estudo continha atributos desnecessários, o que por si só traz alguns inconvenientes (aumento do espaço necessário à persistência do modelo e consequente aumento de custos, maior tempo de resposta dos serviços da plataforma, devido a uma necessidade de trabalhar com dados desnecessários, entre outros).

A utilização desta ferramenta oferece agora à equipa de desenvolvimento uma melhor análise dos modelos existentes na plataforma, permitindo aumentar deste modo a qualidade dos mesmos.

6 Evolução dos modelos

A evolução dos modelos ao longo do tempo é um tema de grande importância, pois é necessário que o modelo seja validado sempre que existam alterações, de forma a nunca existirem incoerências nas aplicações.

Neste capítulo é apresentado um levantamento dessas alterações, em que para cada uma consta o problema que a mesma pode trazer às aplicações, bem como uma proposta de solução. Atualmente, não existe na plataforma myMIS nenhuma validação para qualquer um dos casos apresentados neste capítulo, sendo desejável que as mesmas sejam implementadas no futuro.

Será ainda descrito neste capítulo o problema da gestão de dados que vão ficando nos modelos ao longo do tempo e que não têm qualquer utilidade para as aplicações existentes.

6.1 Alterações no modelo

De forma a evoluir a plataforma myMIS, é necessário implementar na ferramenta de desenvolvimento de modelos a verificação de cada uma destas modificações, bem como dar a possibilidade ao desenvolvedor do modelo de aplicar a solução que pretende, por forma a resolver cada um dos problemas.

De seguida, são apresentadas as alterações que, caso sejam realizadas, levarão a inconsistências nos modelos desenvolvidos e, consequentemente, nas aplicações presentes no sistema.

6.1.1 Atributo

Alteração da propriedade Type

Problema: A alteração desta propriedade pode ter uma das seguintes configurações:

1. Alterar entre tipos de dados base;
2. Alterar entre tipos de dados complexos;
3. Alterar de um tipo de dados base para um complexo;
4. Alterar de um tipo complexo para um tipo de dados base.

Em 1, caso os tipos de dados sejam convertíveis entre si (de um valor numérico para texto, por exemplo) a alteração não representa um problema. Caso não exista a possibilidade de conversão (de uma data para um valor numérico, por exemplo), a propriedade não deve ser alterada, pois os valores já existentes na aplicação não estariam em conformidade com o modelo. O mesmo acontece em 2, 3 e 4: as alterações são entre tipos de dados não convertíveis, representando assim o mesmo problema enunciado anteriormente.

Solução: De forma a solucionar esta questão, existem duas opções a considerar: não permitir alterar a propriedade Type quando a aplicação já contém dados para esse Attribute e não existe conversão possível; ou permitir a alteração da propriedade, sendo definido um valor a colocar em todos os registos existentes na aplicação.

Alteração do valor da propriedade Required para 'verdadeiro'

Problema: Todos os registos existentes do tipo de entidade que contém o Attribute poderão ter um valor de preenchimento obrigatório sem qualquer valor atribuído. Esta situação pode tornar-se um problema mais complexo se o Attribute em questão tiver a propriedade ReadOnly com o valor 'verdadeiro' e não existir uma forma de obter o seu valor (através de uma fórmula, por exemplo), pois assim o Attribute será obrigatório e, por outro lado, o utilizador não o pode preencher.

Solução: Definir um valor a atribuir a cada registo sem valor atribuído.

Adicionar novo em que o valor da propriedade Required é 'verdadeiro'

Problema: Este novo Attribute sendo obrigatório passará a estar presente em todas as instâncias do Type que contém o Attribute em questão. Assim sendo, o modelo irá conter um Attribute obrigatório, cujo valor não existe. Esta situação gera um problema, no sentido em que esse valor pode ser utilizado para uma qualquer ação necessária ao bom funcionamento da aplicação (por exemplo, a exportação de dados para o ERP).

Solução: Definir um valor a atribuir a cada registo já existente e sem valor atribuído.

Remoção quando é utilizado para o cálculo do valor de outro Attribute

Problema: A remoção deste Attribute trará problemas ao modelo, no sentido em que existe um outro Attribute que depende do primeiro para obter o seu valor, seja através de uma fórmula

ou de um Trigger. Desta forma, caso não exista outro modo de obter o valor do segundo atributo e se não for possível alterar o seu valor (propriedade ReadOnly com o valor 'verdadeiro'), a presença deste campo em novos registos passa a ser desnecessária.

Solução: Não permitir remover o Attribute, a não ser que os campos que dependem dele tenham uma outra forma de obter o seu valor, seja através de uma nova fórmula, da definição de um Trigger num outro Attribute ou através da propriedade ReadOnly (que deve ter o valor 'falso').

Alteração de fórmulas

Problema: A alteração de fórmulas pode fazer com que os valores já existentes sejam incorretos de acordo com a nova forma de cálculo. Deste modo, é necessário ao desenvolvedor do modelo perceber se este problema é crítico para o sistema em questão pois, caso seja, a alteração irá fazer com que os dados existentes sejam incoerentes com os novos.

Solução: A resolução deste problema passa por, caso o desenvolvedor do modelo o queira, recalcular todos os registos já existentes.

Alteração de Triggers

Problema: A alteração dos Triggers de um atributo pode fazer com que outros atributos, que dependem do resultado do processamento dos elementos alterados, fiquem sem a possibilidade de que o seu valor seja obtido. Por exemplo, no caso em que o valor de um atributo apenas resulta do processamento de um Trigger de outro atributo, ao ser removido o Trigger, deixa de existir a possibilidade de conhecer o valor do primeiro atributo.

Solução: Todos os atributos afetados pelas alterações devem ser revistos e alterados, caso exista essa necessidade, de forma a ser possível obter o seu valor corretamente.

6.1.2 Sistema de contabilidade

Problema: De forma a manter os dados do sistema de contabilidade corretos e coerentes ao longo do tempo, é essencial controlar de forma muito rigorosa as alterações que podem ser feitas no desenvolvimento deste mecanismo. Qualquer alteração nos Accounting Triggers ou nos seus respetivos Accounting Actions pode tornar os dados existentes até ao momento completamente errados do ponto de vista da nova configuração.

Solução: Existem, para este caso, duas alternativas: uma que permite atualizar a informação já existente e outra que torna a nova informação incompatível com a anterior. Na primeira situação, devem-se atualizar todos os movimentos contabilísticos realizados anteriormente, mas de acordo com a nova configuração dos mecanismos de contabilidade. A segunda opção tornaria a informação do sistema incoerente com a nova configuração, pois a informação já existente não seria atualizada e apenas os novos dados inseridos na aplicação utilizariam a configuração nova dos mecanismos de contabilidade.

6.2 Gestão de informação irrelevante

Ao longo do tempo e das modificações realizadas aos modelos surgem, por vezes, dados que deixam de ter qualquer utilidade para a aplicação em causa. As origens dessa informação inútil podem ser bugs na ferramenta de desenvolvimento ou erros no desenvolvimento dos modelos.

Esta informação irrelevante aumenta os custos necessários ao funcionamento das aplicações, porque para além de essa informação ocupar espaço de armazenamento nas estruturas de persistência de dados, a quantidade de informação a circular é maior e aumenta também o tempo de processamento dos serviços na plataforma, seja na obtenção ou na gravação de dados.

Assim, a identificação desta informação tem uma grande importância, pois a sua remoção permite aumentar a performance das aplicações criadas e baixar custos de manutenção da plataforma.

7 Conclusões

Através do trabalho realizado foi possível solucionar os dois problemas a que esta tese pretendia dar resposta: a inexistência de formalização da linguagem BAMoL e a falta de mecanismos de validação de modelos.

Num primeiro momento, foi concretizada a formalização de uma linguagem de domínio específico, a BAMoL, através de uma descrição textual de todos os seus elementos e da criação de uma gramática representativa da mesma.

A descrição textual da BAMoL que foi criada, tem um papel importante para o desenvolvimento de modelos por parte de novos desenvolvedores, no sentido em que lhes permite conhecer e perceber quais os conceitos da linguagem e para que servem, bem como as relações entre os mesmos. Desta forma, o processo de educação de novos desenvolvedores de BAMoL fica mais simplificado do que anteriormente, pois agora passa a existir um documento explicativo da linguagem em toda a sua extensão.

Apesar de esta descrição permitir uma melhor educação de novos desenvolvedores de BAMoL, a mesma pode encontrar-se produzida de uma forma muito técnica, isto é, com um discurso muito próximo daquilo que os desenvolvedores internos da plataforma conhecem e mais distante da linguagem dos especialistas do domínio que pretendam implementar as suas soluções. Desta forma, pode gerar-se alguma resistência à perceção dos conceitos, que consequentemente pode levar a um abandono da linguagem ou a implementações realizadas de uma forma errada.

A gramática concebida é utilizada posteriormente num dos outros assuntos a que esta tese pretende dar resposta: a validação de modelos criados para a plataforma myMIS. Em resposta a este problema, foi desenhada e desenvolvida uma ferramenta que permite auxiliar a equipa de desenvolvimento na validação dos modelos, melhorando assim o controlo dos modelos existentes no sistema.

Tendo como objetivo validar os resultados obtidos através da ferramenta de validação de modelos criada no âmbito desta tese, foi realizada uma atividade experimental. Utilizando um modelo existente, atualmente em produção, foi possível encontrar dados incorretos nesse mesmo modelo. Desta forma, foi possível aferir que a ferramenta permite encontrar erros nos modelos da plataforma, validando assim o propósito para o qual foi criada.

Com o último ponto realizado nesta tese, a equipa de desenvolvimento da plataforma myMIS ganhou uma ferramenta útil para detetar erros nos modelos criados com a BAMoL. Isso permite criar aplicações mais eficientes (devido à inexistência de propriedades irrelevantes às mesmas), bem como diminuir os custos de manutenção de todo o sistema (pelo facto de ser reduzida a informação guardada nas estruturas de persistências de dados e que circula na plataforma).

Apesar de a ferramenta desenvolvida atualmente permitir validar os modelos, a mesma necessita que sejam adicionadas mais algumas validações, pois ainda contém algumas limitações nesse sentido, o que faz com que os modelos não sejam validados na sua totalidade.

De forma a manter as soluções apresentadas com a utilidade pretendida, é necessário que as evoluções da plataforma sejam reproduzidas tanto na descrição textual da linguagem como na ferramenta de validação de modelos. Neste último caso, para além da própria aplicação, é indispensável a atualização da gramática da linguagem, bem como das validações que recorrem ao uso do XSLT.

Utilizando o mecanismo de validação de modelos criado no âmbito desta tese, é desejável que exista a possibilidade na ferramenta de desenvolvimento de validar os modelos desenvolvidos na mesma. Para tal, apenas é necessário enquadrar na ferramenta de desenvolvimento de modelos, a ferramenta de validação criada e explicada neste documento.

Como foi referido anteriormente, a forma de guardar os modelos localmente é feita recorrendo a um conjunto de ficheiros ilegíveis. Recorrendo aos mecanismos desenvolvidos nesta tese para a conversão de um modelo no formato XML, é desejável que seja adicionada a possibilidade de exportação dos modelos para esse mesmo formato. Essa opção deve ser incluída na ferramenta de desenvolvimento de modelos.

Referências

[Atkinson, C. *et al.*, 2003] Atkinson, C. *et al.* - Model-Driven Development: A Metamodeling Foundation, 2003. https://spemarti.googlecode.com/files/Atkinson2003.pdf [último acesso: Jan 2014]

[Deursen, A. *et al.*, 2000] Deursen, A. *et al.* - Domain-Specific Languages: An Annotated Bibliography, 2000. http://www.st.ewi.tudelft.nl/~arie/papers/dslbib.pdf [último acesso: Set 2014]

[Fowler, M, 2010] Fowler, M. - Domain Specific Languages. Addison-Wesley Professional, 2010

[France, R. *et al.*] France, R. *et al.* - Model-driven Development of Complex Software: A Research Roadmap. http://sse-tubs.de/publications/FR_MDDofComplexSoftware_ICSE_07.pdf [último acesso: Jan 2014]

[Hruby, P. *et al.*] Hruby, P. *et al.* - Model-Driven Design Using Business Patterns (Chapter What is REA). http://www.springer.com/cda/content/document/cda_downloaddocument/9783540301547-c1.pdf?SGWID=0-0-45-295789-p96990473 [último acesso: Jan 2014]

[Hudak, P., 1997] Hudak, P. - Domain Specific Languages, 1997. http://haskell.cs.yale.edu/wp-content/uploads/2011/01/DSEL-Little.pdf [último acesso: Set 2014]

[McCarthy, W., 1982] McCarthy, W. – The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment, 1982. https://www.msu.edu/~mccarth4/McCarthy.pdf [último acesso: Jan 2014]

[Mellor, S. *et al.*, 2003] Mellor, S. *et al.* - Model-Driven Development, 2003. https://eprints.mdx.ac.uk/6083/1/s5014.pdf [último acesso: Jan 2014]

[Selic, B., 2003] Selic, B. - The Pragmatics of Model-Driven Development, 2003. http://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/papers/SelicPragmatics.pdf [último acesso: Jan 2014]

[Sendall, S. *et al.*, 2003] Sendall, S. *et al.* - Model Transformation: The Heart and Soul of Model-Driven Software Development. http://spectral.mscs.mu.edu/ASD2007/lectures/mdd-transformation.pdf [último acesso: Jan 2014]

[Voelter, M. *et al.*] Voelter, M. *et al* - Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. http://voelter.de/data/pub/VoelterGroher_SPLEwithAOandMDD.pdf [último acesso: Jan 2014]

[Voelter, M., 2013] Voelter, M - DSL Engineering: Designing, Implementing and Using Domain-Specific Languages, 2013. http://voelter.de/dslbook/markusvoelter-dslengineering-1.0.pdf [último acesso: Out 2013]

[Yoder, J. *et al.* (1)] Yoder, J. *et al.* - The Adaptive Object-Model Architectural Style. http://joeyoder.com/Research/metadata/WICSA3/ArchitectureOfAOMsWICSA3.pdf [último acesso: Jan 2014]

[Yoder, J. *et al.* (2)] Yoder, J. *et al.* - Architecture and Design of Adaptive Object-Models. http://www.itu.dk/courses/VOP/E2008/lessons/12_AOM.pdf [último acesso: Jan 2014]

Anexos

Anexo 1 – Gramática da linguagem BAMoL no formato XML Schema

Modelo

```
<xs:element name="Model">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="AgentTypes">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="EntityType" type="mymis:EntityType"
maxOccurs="unbounded" minOccurs="0"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="ResourceTypes">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="EntityType" type="mymis:EntityType"
maxOccurs="unbounded" minOccurs="0"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="UserDefinedTypes">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="EntityType" type="mymis:EntityType"
maxOccurs="unbounded" minOccurs="0"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="CommitmentTypes">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CommitmentType"
type="mymis:TransactionalEntityType" maxOccurs="unbounded" minOccurs="0"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="EventTypes">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="EventType"
type="mymis:TransactionalEntityType" maxOccurs="unbounded" minOccurs="0"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="ProcessTypes">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ProcessType" type="mymis:ProcessType"
maxOccurs="unbounded" minOccurs="0"/>
```

```
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Agent, Resource e User Defined Type

```
<xs:complexType name="EntityType">
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Attributes">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Attribute" maxOccurs="unbounded"
minOccurs="0" type="mymis:Attribute"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="EntityItems">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="EntityItem" maxOccurs="unbounded"
minOccurs="0" type="mymis:EntityItemType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="ApprovalStages">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Stage" maxOccurs="unbounded" minOccurs="0"
type="mymis:ApprovalStage"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>

      <xs:element name="IsResponsibilityCenterType" type="xs:boolean"/>
      <xs:element name="IsCompanyType" minOccurs="0" maxOccurs="1"
type="xs:boolean"/>
    </xs:sequence>
    <xs:attribute name="Code" type="xs:ID" use="required" />
  </xs:complexType>
```

Commitment e Event Type

```
<xs:complexType name="TransactionalEntityType">
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Kind">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="I" />
            <xs:enumeration value="D" />
          </xs:restriction>
```

```
        </xs:simpleType>
      </xs:element>
      <xs:element name="ActivityKind">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="O" />
            <xs:enumeration value="F" />
            <xs:enumeration value="I" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="ResourceType" type="xs:string"/>
      <xs:element name="ProviderAgentType" type="xs:string"/>
      <xs:element name="ReceiverAgentType" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="Code" type="xs:string" use="required" />
  </xs:complexType>
```

Process Type

```
  <xs:complexType name="ProcessType">
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="InteractionTypes">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="InteractionType" type="mymis:InteractionType"
maxOccurs="unbounded" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="Code" type="xs:string" use="required" />
  </xs:complexType>
```

Interaction Type

```
<xs:complexType name="InteractionType">
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Attributes">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Attribute" maxOccurs="unbounded"
minOccurs="0" type="mymis:Attribute"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="EntityItems">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="EntityItem" maxOccurs="unbounded"
minOccurs="0" type="mymis:EntityItemType"/>
          </xs:sequence>
```

```
        </xs:complexType>
      </xs:element>
      <xs:element name="ApprovalStages">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Stage" maxOccurs="unbounded" minOccurs="0"
type="mymis:ApprovalStage"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Details">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="EntityType" type="xs:string"/>
            <xs:element name="Attributes">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Attribute" maxOccurs="unbounded"
minOccurs="0" type="mymis:Attribute"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="Fulfillments">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Fulfillment" maxOccurs="unbounded"
minOccurs="0" type="mymis:Fulfillment"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="Rectifications">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Rectification" maxOccurs="unbounded"
minOccurs="0" type="mymis:Rectification"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="Taxes">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Tax" maxOccurs="unbounded"
minOccurs="0" type="mymis:Tax"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="PolicyTypes">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="PolicyType" type="mymis:PolicyType"
maxOccurs="unbounded" minOccurs="0"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Summary" minOccurs="0" maxOccurs="1">
        <xs:complexType>
          <xs:sequence>
```

```
            <xs:element name="EntityType" type="xs:string"/>
            <xs:element name="Attributes">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Attribute" maxOccurs="unbounded"
minOccurs="0" type="mymis:Attribute"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="Fulfillments">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Fulfillment" maxOccurs="unbounded"
minOccurs="0" type="mymis:Fulfillment"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="Rectifications">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Rectification" maxOccurs="unbounded"
minOccurs="0" type="mymis:Rectification"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="Taxes">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Tax" maxOccurs="unbounded"
minOccurs="0" type="mymis:Tax"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="PolicyTypes">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="PolicyType" type="mymis:PolicyType"
maxOccurs="unbounded" minOccurs="0"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Pattern">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="AddToPrincipal" />
            <xs:enumeration value="SelectInPrincipal" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="Code" type="xs:string" use="required" />
  </xs:complexType>
```

Entity Item Type

```
<xs:complexType name="EntityItemType">
    <xs:sequence>
      <xs:element name="Code" type="xs:string"/>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Attributes">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Attribute" maxOccurs="unbounded"
minOccurs="0" type="mymis:Attribute"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

Fulfillment

```
<xs:complexType name="Fulfillment">
    <xs:sequence>
      <xs:element name="Kind">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="Equal" />
            <xs:enumeration value="LessThan" />
            <xs:enumeration value="GreaterThan" />
            <xs:enumeration value="Any" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="AgentAttribute" type="xs:string"/>
      <xs:element name="List" type="xs:string"/>
      <xs:element name="Attribute" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="From" type="xs:string" />
            <xs:element name="To" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Condition" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Value" type="xs:string" />
            <xs:element name="Operator">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="Equal" />
                  <xs:enumeration value="Less" />
                  <xs:enumeration value="LessOrEqual" />
                  <xs:enumeration value="Greater" />
                  <xs:enumeration value="GreaterOrEqual" />
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
            <xs:element name="Element" type="xs:string" />
```

```
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
```

Rectification

```
<xs:complexType name="Rectification">
    <xs:sequence>
      <xs:element name="Kind">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="Equal" />
            <xs:enumeration value="LessThan" />
            <xs:enumeration value="GreaterThan" />
            <xs:enumeration value="Any" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="AgentAttribute" type="xs:string"/>
      <xs:element name="List" type="xs:string"/>
      <xs:element name="Attribute" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="From" type="xs:string" />
            <xs:element name="To" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Condition" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Value" type="xs:string" />
            <xs:element name="Operator">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="Equal" />
                  <xs:enumeration value="Less" />
                  <xs:enumeration value="LessOrEqual" />
                  <xs:enumeration value="Greater" />
                  <xs:enumeration value="GreaterOrEqual" />
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
            <xs:element name="Element" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

Tax

```
<xs:complexType name="Tax">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="EntityType" type="xs:IDREF"/>
    <xs:element name="ResultType" type="xs:IDREF"/>
    <xs:element name="Attribute" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="ToGet" type="xs:string" />
          <xs:element name="ToSet" type="xs:string" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Approval Stage

```
<xs:complexType name="ApprovalStage">
    <xs:sequence>
      <xs:element name="Code" type="xs:string"/>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="IsStart" type="xs:boolean" />
      <xs:element name="IsEnd" type="xs:boolean" />
      <xs:element name="EvaluateExecution" type="xs:string"/>
      <xs:element name="WhenApproved" type="xs:string"/>
      <xs:element name="WhenRejected" type="xs:string"/>
      <xs:element name="EditableAttributes">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Attribute" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="ApproverRules">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Rule" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="EvaluateExecution" type="xs:string"/>
                  <xs:element name="AssignTo" type="xs:string"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

Policy Type

```
<xs:complexType name="PolicyType">
    <xs:sequence>
      <xs:element name="Code" type="xs:string"/>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Attributes">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Attribute">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Code" type="xs:string"/>
                  <xs:element name="Operator">
                    <xs:simpleType>
                      <xs:restriction base="xs:string">
                        <xs:enumeration value="Equal" />
                        <xs:enumeration value="Less" />
                        <xs:enumeration value="LessOrEqual" />
                        <xs:enumeration value="Greater" />
                        <xs:enumeration value="GreaterOrEqual" />
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Account">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Period">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="Day" />
                  <xs:enumeration value="Month" />
                  <xs:enumeration value="Year" />
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
            <xs:element name="AccountName" type="xs:string"/>
            <xs:element name="Operator">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="Equal" />
                  <xs:enumeration value="Less" />
                  <xs:enumeration value="LessOrEqual" />
                  <xs:enumeration value="Greater" />
                  <xs:enumeration value="GreaterOrEqual" />
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="ResponsibilityCenter" minOccurs="0"
maxOccurs="unbounded">
```

```
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Code" type="xs:string"/>
            <xs:element name="ValueToCompare" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

Attribute

```
  <xs:complexType name="Attribute">
      <xs:sequence>
        <xs:element name="Code" type="xs:string"/>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Description" type="xs:string"/>
        <xs:element name="Type">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="ST1" />
              <xs:enumeration value="ST2" />
              <xs:enumeration value="IN1" />
              <xs:enumeration value="DT1" />
              <xs:enumeration value="DT2" />
              <xs:enumeration value="BO1" />
              <xs:enumeration value="BD1" />
              <xs:enumeration value="BD2" />
              <xs:enumeration value="DE1" />
              <xs:enumeration value="DE2" />
              <xs:enumeration value="DE3" />
              <xs:enumeration value="PS1" />
              <xs:enumeration value="CT" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="ComplexType" minOccurs="0" maxOccurs="1"
  type="mymis:ComplexType" />
        <xs:element name="Required" type="xs:boolean"/>
        <xs:element name="RequiredCondition" type="mymis:Formula"/>
        <xs:element name="ExistenceCondition" type="mymis:Formula"/>
        <xs:element name="Min" type="xs:double" nillable="true"/>
        <xs:element name="Max" type="xs:double" nillable="true"/>
        <xs:element name="DefaultValue" type="xs:string" nillable="true"/>
        <xs:element name="Editable" type="xs:boolean"/>
        <xs:element name="Formula" type="mymis:Formula"/>
        <xs:element name="ValidationFormula" type="mymis:Formula"/>
        <xs:element name="AllowMultipleFiles" type="xs:boolean" minOccurs="0"
  maxOccurs="1"/>
        <xs:element name="AllowFileTypes" type="xs:string" minOccurs="0"
  maxOccurs="1"/>
        <xs:element name="ReadOnly" type="xs:boolean"/>
        <xs:element name="ReadOnlyCondition" type="mymis:Formula"/>
        <xs:element name="Visible" type="xs:boolean"/>
        <xs:element name="VisibleCondition" type="mymis:Formula"/>
        <xs:element name="Triggers">
          <xs:complexType>
```

```
          <xs:sequence>
            <xs:element name="Trigger" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Name" type="xs:string"/>
                  <xs:element name="Type">
                    <xs:simpleType>
                      <xs:restriction base="xs:string">
                        <xs:enumeration value="CRT" />
                        <xs:enumeration value="CHG" />
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:element>
                  <xs:element name="Actions">
                    <xs:complexType>
                      <xs:sequence>
                        <xs:element name="Action" maxOccurs="unbounded"
minOccurs="0">
                          <xs:complexType>
                            <xs:sequence>
                              <xs:element name="Type">
                                <xs:simpleType>
                                  <xs:restriction base="xs:string">
                                    <xs:enumeration value="GetUserAgent" />
                                    <xs:enumeration value="GetNumber" />
                                    <xs:enumeration value="GetEntity" />
                                    <xs:enumeration value="GetEntityItem"
/>
                                    <xs:enumeration
value="GetEntityMessage" />
                                    <xs:enumeration value="GetAccounts" />
                                    <xs:enumeration
value="GetTransactionalEntities" />
                                    <xs:enumeration value="Formula" />
                                    <xs:enumeration value="Check" />
                                  </xs:restriction>
                                </xs:simpleType>
                              </xs:element>
                              <xs:element name="EntityKind">
                                <xs:simpleType>
                                  <xs:restriction base="xs:string">
                                    <xs:enumeration value="Agent" />
                                    <xs:enumeration value="Resource" />
                                    <xs:enumeration
value="UserDefinedEntity" />
                                    <xs:enumeration value="Commitment" />
                                    <xs:enumeration value="Event" />
                                    <xs:enumeration value="Interaction" />
                                    <xs:enumeration value="" />
                                  </xs:restriction>
                                </xs:simpleType>
                              </xs:element>
                              <xs:element name="EntityType"
type="xs:string" />
                              <xs:element name="Attributes">
                                <xs:complexType>
                                  <xs:sequence>
                                    <xs:element name="AttributePair"
minOccurs="0" maxOccurs="unbounded">
```

```
                                        <xs:complexType>
                                          <xs:sequence>
                                            <xs:element name="ToGet"
type="xs:string" />
                                            <xs:element name="ToSet"
type="xs:string" />
                                          </xs:sequence>
                                        </xs:complexType>
                                      </xs:element>
                                    </xs:sequence>
                                  </xs:complexType>
                                </xs:element>
                                <xs:element name="Conditions">
                                  <xs:complexType>
                                    <xs:sequence>
                                      <xs:element name="Condition"
minOccurs="0" maxOccurs="unbounded">
                                        <xs:complexType>
                                          <xs:sequence>
                                            <xs:element name="Value"
type="xs:string" />
                                            <xs:element name="Operator">
                                              <xs:simpleType>
                                                <xs:restriction
base="xs:string">
                                                  <xs:enumeration value="="
/>
                                                  <xs:enumeration
value="&lt;" />
                                                  <xs:enumeration
value="&lt;=" />
                                                  <xs:enumeration
value="&gt;" />
                                                  <xs:enumeration
value="&gt;=" />
                                                  <xs:enumeration value="" />
                                                </xs:restriction>
                                              </xs:simpleType>
                                            </xs:element>
                                            <xs:element name="Attribute"
type="xs:string" />
                                          </xs:sequence>
                                        </xs:complexType>
                                      </xs:element>
                                    </xs:sequence>
                                  </xs:complexType>
                                </xs:element>
                              </xs:sequence>
                            </xs:complexType>
                          </xs:element>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
```

```
  </xs:complexType>


  <xs:complexType name="Formula">
    <xs:sequence>
      <xs:element name="Elements" type="mymis:FormulaElements" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="FormulaElements">
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Element" type="mymis:FormulaElement"
maxOccurs="unbounded"/>
        <xs:element name="Operator" type="mymis:FormulaOperator"
maxOccurs="unbounded"/>
        <xs:element name="Function" type="mymis:FormulaFunction"
maxOccurs="unbounded"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="FormulaElement">
    <xs:sequence>
      <xs:element name="Location" type="xs:string" />
      <xs:element name="Value" type="xs:string" nillable="true"
minOccurs="0" maxOccurs="1" />
      <xs:element name="Elements" type="mymis:FormulaElements" />
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="FormulaOperator">
    <xs:restriction base="xs:string">
      <xs:enumeration value="+" />
      <xs:enumeration value="-" />
      <xs:enumeration value="*" />
      <xs:enumeration value="/" />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="FormulaFunction">
    <xs:sequence>
      <xs:element name="Code">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="SUM" />
            <xs:enumeration value="SUMIFS" />
            <xs:enumeration value="TODAY" />
            <xs:enumeration value="IF" />
            <xs:enumeration value="CONCATENATE" />
            <xs:enumeration value="DAY" />
            <xs:enumeration value="MONTH" />
            <xs:enumeration value="YEAR" />
            <xs:enumeration value="DATE" />
            <xs:enumeration value="ValidaNIF" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
```

```
      <xs:element name="Parameters">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Element" type="mymis:FormulaElement"
maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>


  <xs:complexType name="ComplexType">
    <xs:sequence>
      <xs:element name="Kind">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="Agent" />
            <xs:enumeration value="Resource" />
            <xs:enumeration value="UserDefinedEntity" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="EntityType" type="xs:IDREF"></xs:element>
      <xs:element name="Cardinality" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
```