Um Metamodelo para Programas Adaptativos Utilizando SBMM

[1]S. R. M. Canovas, [2]C. E. Cugnasca

***Abstract* — *Model Driven Architecture* (MDA) é uma abordagem para desenvolvimento de software baseada na transformação sucessiva de modelos de alto nível até a geração do código-fonte. Utiliza tecnologias específicas mantidas pelo *Object Management Group* (OMG). Uma delas é o *Meta Object Facility* (MOF), que permite descrever metamodelos, ou seja, modelos das linguagens de modelagem de software tais como a UML. Os metamodelos são elementos chave para a descrição de transformações de modelos, base da MDA. O *Set Based Meta Modeling* (SBMM) é um formalismo que se propõe a ser uma alternativa mais simples em relação ao MOF e que, portanto, também permite descrever metamodelos. Este trabalho apresenta um metamodelo para programas adaptativos definido em SBMM. Este metamodelo descreve uma linguagem gráfica encontrada na literatura para representar programas adaptativos. Com isso, podem ser aplicadas técnicas, conceitos e ferramentas da MDA para a geração automática de código adaptativo a partir de modelos de mais alto nível.**

***Keywords*— Tecnologias adaptativas (*adaptive technologies*), programas adaptativos (*adaptive programs*), *Model Driven Architecture, Set Based Meta Modeling*.**

I. INTRODUÇÃO

MODEL Driven Architecture (MDA) é uma abordagem para desenvolvimento de software através da qual um software é construído através da transformação sucessiva e encadeada de modelos. Um modelo de um sistema é uma descrição ou especificação do mesmo considerando um determinado propósito. Usualmente um modelo consiste em uma combinação de textos e desenhos, podendo estar descrito em linguagem de modelagem (ex: UML) ou em linguagem natural [1].

Em sua forma mais básica, a MDA propõe dois modelos: o *platform-independent model* (PIM), totalmente independente de plataforma, e o *platform-specific model* (PSM), dependente de plataforma [2]. A MDA propõe que o PSM seja gerado a partir do PIM através de uma transformação de modelos. Uma transformação é o processo de converter um modelo em outro modelo do mesmo sistema, conforme ilustrado na Figura 1.

Esta figura ilustra o esquema básico de uma transformação segundo a MDA: um PIM, junto com informações adicionais (quadro em branco), passa por um mecanismo de transformação para gerar um PSM.

As informações adicionais representadas pelo quadro em branco podem ser tanto para utilização do mecanismo de transformação (ex: mapeamentos, regras, templates, padrões, procedimentos) quanto informações extras para complementar o PIM e prover informações suficientes para gerar o PSM (ex: marcações). Isso se faz necessário, pois como o PIM não considera aspectos específicos da plataforma para a qual o sistema será gerado, mas o PSM considera, é preciso introduzir informações extras para cobrir os novos detalhes que aparecem.

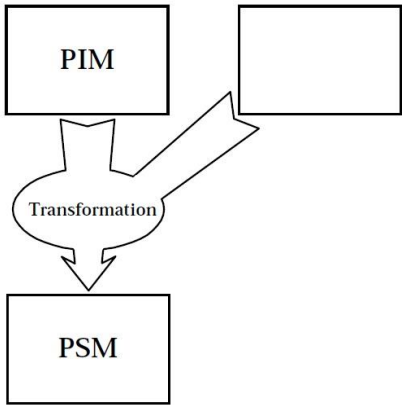

Fig. 1. Transformação de modelos segundo a MDA [1].

Uma próxima etapa de transformação a partir do PSM seria para a geração do código-fonte do sistema desejado. Este tipo de transformação é chamada de modelo-para-código (*model-to-code transformation*). Porém, se abstrairmos o conceito e considerarmos que o código-fonte também é um tipo de modelo textual, descrito em uma linguagem de programação, então esta nomenclatura tem pouca importância.

A MDA propõe diversos mecanismos para que a transformação seja realizada [1], podendo até mesmo ser manual. Porém, é claro que se deseja o máximo de automação possível no processo através da utilização de ferramentas de software, de forma que na situação ideal o PIM é suficiente para descrever o sistema por completo mas ao mesmo tempo se manter independente de plataforma. As informações adicionais necessárias para a transformação já estariam pré-definidas na forma de conjuntos de *templates*, regras e padrões disponíveis para aplicação no processo. Assim, implementações do mesmo sistema podem ser geradas automaticamente para diversas plataformas através de transformações específicas para diversos PSMs a partir do mesmo PIM.

Na situação ideal, qualquer manutenção no sistema se reduz a uma manutenção no PIM. Através da reaplicação da transformação com as informações adicionais já definidas, as implementações do sistema em cada plataforma são regeradas.

Desta forma, os arquitetos e desenvolvedores de software podem focar seus esforços em desenvolver e manter um

[1] S. R. M. Canovas, Escola Politécnica da Universidade de São Paulo, Brasil (e-mail: sergio.canovas@usp.br).

[2] C. E. Cugnasca, Escola Politécnica da Universidade de São Paulo, Brasil (e-mail: carlos.cugnasca@gmail.com).

modelo conceitual abstrato da aplicação (PIM), que contém as regras de negócio e seus aspectos fundamentais. A obtenção de uma implementação para uma plataforma específica ocorreria rapidamente e de forma automatizada. Os investimentos das organizações em projetos de software seriam então preservados na medida em que um mesmo PIM, já criado, poderia dar origem automaticamente a novas implementações para outras plataformas que surgem no contexto da organização [3]. Mais interessante ainda, esta arquitetura permite gerar sistemas existentes através de seu PIM para novas plataformas que ainda estão por ser inventadas, sem necessidade de desenvolver novamente o sistema para essas plataformas.

A MDA não define plataforma, linguagem de programação ou *middleware* específicos. É um conceito abstrato dentro do qual se encaixam as ferramentas específicas. Atualmente a MDA é implementada por diversos fabricantes através de ferramentas que operam em cadeia para atingir os objetivos [4]. A MDA é mantida pela *Object Management Group* (OMG) [5].

A MDA possui como objetivos prover três benefícios principais através da separação dos modelos [1]:

- Portabilidade;
- Interoperabilidade;
- Reusabilidade.

Para a MDA funcionar na prática com transformações automáticas, é necessário especificar funções de mapeamento. Funções de mapeamento são descritas sobre metamodelos.

O *Set Based Meta Modeling* é um formalismo desenvolvido como parte desta pesquisa que se propõe a descrever metamodelos utilizando conceitos matemáticos fundamentais: conjuntos, relações e funções.

O objetivo deste trabalho é apresentar um metamodelo definido em SBMM que descreve uma linguagem gráfica de representação de programas adaptativos definida em [6]. A definição deste metamodelo é um passo para a criação de uma ferramenta CASE para programas adaptativos, ou então para a utilização de ferramentas meta-CASE. Ferramentas meta-CASE são capazes de gerar ferramentas CASE customizadas [14]. Metamodelos são entradas para essas ferramentas. Como visto anteriormente, outra importância do metamodelo é permitir especificar funções de mapeamento para transformações automáticas, inclusive geração de código-fonte.

Embora não faça parte do escopo deste trabalho abordar as funções de mapeamento, a existência do metamodelo formal viabiliza a escrita das mesmas, sendo outro passo importante para a geração automática de programas adaptativos a partir de um PIM, ou seja, para a aplicação da MDA na construção de programas adaptativos.

A seção II introduz a chamada arquitetura das quatro camadas, uma relação conceitual entre níveis ou camadas descritivas que faz parte do arcabouço teórico da MDA. A seção III apresenta a linguagem gráfica para representação de programas adaptativos definida por [6]. A seção IV descreve o *Set Based Meta Modeling*, formalismo para definição de metamodelos a ser utilizado, e aplica o SBMM para descrever o metamodelo proposto. A seção V discute os resultados, enquanto a seção VI apresenta as conclusões. Por fim, a seção VII lista as referências bibilográficas.

II. A Arquitetura das Quatro Camadas

A arquitetura das quatro camadas [15] é um modelo conceitual que contempla objetos de tempo de execução, modelos, metamodelos e um último nível conhecido informalmente como metametamodelo, conforme pode ser visto na Figura 2.

Muitas vezes fala-se sobre instâncias, classes, modelos, metamodelos, etc. de forma conjunta. Esta arquitetura permite classificar e visualizar precisamente onde cada elemento se encontra, bem como identificar suas relações horizontais e verticais, facilitando a leitura e tratamento desses elementos por máquinas.

A camada M0 contém os dados de um sistema em tempo de execução. No exemplo do sistema escolar, seriam objetos instanciados de CAluno, CCurso, etc. durante a execução do software. Registros em tabelas de um banco de dados relacional que armazenam esses objetos de forma persistente também pertencem a esta camada.

A camada M1 contém os modelos do sistema considerado: diagramas UML de classes, definições de tabelas de um banco de dados relacional, diagramas de casos de uso, códigos-fonte, etc. É o nível onde a modelagem do software ocorre e, portanto, onde trabalham os analistas e desenvolvedores. As transformações de modelos da Figura 1 ocorrem dentro deste nível.

A camada M2 contém os metamodelos, isto é, os metadados que capturam as linguagens de modelagem. Neste nível estão as definições da UML, ou seja, as especificações de seus diagramas e elementos: classes, atributos, operações, casos de uso, atores, etc. Definições sintáticas e semânticas das linguagens de programação também ocorrem nesse nível. Por isso, é aqui que atuam os desenvolvedores de linguagens. As funções de mapeamento devem ser definidas dentro deste nível, entre os metamodelos. Desta forma, como vimos, passam a ter utilidade geral pois podem ser aplicadas sobre instâncias quaisquer desses metamodelos (ou seja, os modelos de software do nível M1).

A camada M3 contém os "metametamodelos" que servem para descrever os metamodelos. Este nível contém apenas os conceitos mais simples requeridos para capturar modelos e metamodelos, sendo uma constante para suportar todas as possibilidades de modelagem das camadas acima. A OMG dedica esforços de padronização neste nível, e com isso criou o MOF [7]. O SBMM é uma alternativa ao MOF.

Uma confusão comum que se faz à primeira vista é tentar enxergar as transformações da MDA ocorrendo entre as camadas. Na verdade, as transformações da MDA ocorrem dentro da mesma camada. Em particular, as transformações de interesse para desenvolvimento de software (Figura 1) ocorrem dentro da camada M1. A relação entre as camadas é de descrição. O nível M3 possui elementos para descrever

metamodelos do nível M2, que por sua vez permitem descrever modelos do nível M1, que por sua vez permitem descrever os programas executáveis do nível M0.

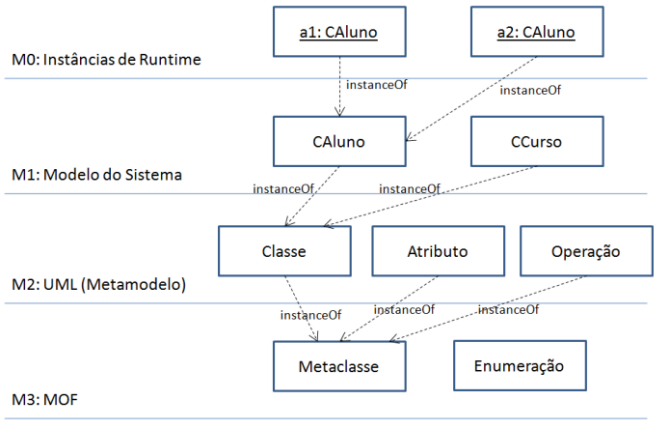

Fig. 2. A arquitetura das quatro camadas

Foi visto, e cabe ressaltar, que a utilidade prática da arquitetura das quatro camadas para a MDA está no fato de que cada camada serve para prover conceitos que permitem descrever instâncias da camada anterior. Por exemplo, os modelos da camada M1 descrevem programas cuja execução está na camada M0. Os metamodelos da camada M2 descrevem as linguagens de programação ou de modelagem que existem na camada M1.

Como as funções de mapeamento devem ser definidas sobre metamodelos, então a existência de conceitos e blocos básicos comuns na camada M3 (que são utilizados para definir diferentes metamodelos da camada M2) permitiria a definição de funções de mapeamento entre metamodelos através de técnicas comuns. Se surgirem novos metamodelos, ou seja, novas linguagens de modelagem, as mesmas técnicas ainda poderiam ser usadas para criar funções de mapeamento que atuam sobre modelos escritos nessas novas linguagens de modelagem. Daí a importância de existir uma especificação de linguagem para a camada M3 com sintaxe e semântica suficientemente simples para ser implementada e utilizada, mas ao mesmo tempo poderosa para permitir que novos metamodelos surjam conforme necessidades práticas.

III. Programas Adaptativos e Representação Gráfica

Um dispositivo adaptativo é composto por um dispositivo não-adaptativo subjacente e um mecanismo formado por funções adaptativas capaz de alterar o conjunto de regras que define seu comportamento [11]. Esta camada adaptativa confere ao dispositivo capacidade de automodificação, onde as alterações nas regras de comportamento são disparadas em função da configuração corrente do dispositivo e dos estímulos recebidos. Essas alterações se caracterizam pela substituição, inserção ou remoção de regras.

Um autômato adaptativo, por exemplo, é um dispositivo adaptativo que estende o conceito de autômato finito incorporando a característica de desenvolver uma autoreconfiguração em resposta a um estímulo externo [12]. Criação de novos estados e transições em tempo de execução são exemplos de autoreconfiguração.

Um programa adaptativo pode ser entendido como uma especificação de uma sequência automodificável de instruções, que representa um código dinamicamente alterável [6]. Podem ser considerados dispositivos adaptativos onde o dispositivo não-adaptativo subjacente seria um programa estático.

Em um programa adaptativo, as ações adaptativas podem inserir ou remover linhas de código, antes ou depois de processar um estímulo.

Basic Adaptive Language (BADAL) é uma linguagem de programação adaptativa de alto nível proposta por [6]. Uma linguagem adaptativa deve prover instruções explícitas para alteração do código-fonte em tempo de execução, e assim o faz a BADAL.

O compilador BADAL apresentado por [6] gera código para o ambiente de execução desenvolvido em [13], que é uma máquina virtual com características específicas que possibilita que um programa realize automodificações em seu código em tempo de execução.

Juntamente com a linguagem BADAL, uma representação gráfica para programas adaptativos é apresentada em [6], descrita em linguagem natural. Cada instância dessa representação gráfica é um modelo, não necessariamente completo, para um programa adaptativo, assim como um diagrama de classes UML é um modelo, também não necessariamente completo, para um programa orientado a objetos.

Se for possível criar um metamodelo formal para essa representação gráfica de programas adaptativos, assim como existem metamodelos formais para os modelos UML, viabiliza-se a aplicação da MDA para a criação de código adaptativo a partir de PIMs e transformação de modelos, já que se estabelece a base para a definição de funções de mapeamento.

Desta maneira, podemos ter programas adaptativos descritos em um modelo com representação em alto nível, mesmo que parcialmente, e, futuramente, o mesmo modelo poderá ser usado para gerar código adaptativo para outras eventuais linguagens adaptativas e plataformas de execução de software adaptativo.

No restante desta seção descreveremos em linguagem natural a representação gráfica proposta por [6], para mais adiante criar um metamodelo formal.

Primeiramente será apresentada uma notação para o dispositivo subjacente não-adaptativo do programa adaptativo, isto é, o programa estático escrito em uma linguagem de alto nível hospedeira. A Figura 3 [6] apresenta a arquitetura de um programa projetado como um dispositivo guiado por regras, onde é possível verificar uma camada de código formado por blocos básicos escritos em linguagem hospedeira (camada 1).

A camada 3 provê conexões que ligam a saída de um bloco básico à entrada de algum dos blocos básicos do programa. O valor de saída do bloco recém executado é utilizado por um decisor (camada 2), que encaminha a execução do programa para um dos blocos conectados à saída em função deste valor.

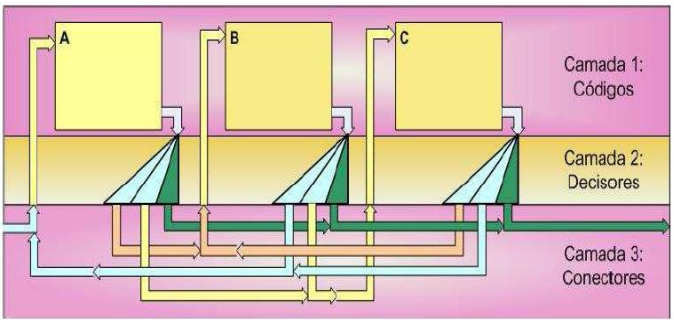

Fig. 3. Arquitetura de um programa não-adaptativo na forma de um dispositivo guiado por regras [6].

Define-se um bloco básico como sendo uma parcela do programa expressa na forma de uma sequência de comandos da linguagem hospedeira, e que deve ser descrito de tal modo que apresente uma só entrada e uma só saída. O valor de saída deve exprimir de alguma forma convencionada uma condição referente ao resultado de sua execução.

Desta forma, os programas adaptativos a serem elaborados pelo programador contêm, como elementos construtivos iniciais, blocos básicos escritos puramente na linguagem hospedeira [6]. Para assegurar a coerência estrutural dos programas assim construídos, é preciso que o programador projete adequadamente as conexões e decisores, e que o compilador faça as validações necessárias.

Cabe ao programador definir os possíveis valores de saída de cada bloco básico. Caso se obtenha um valor de saída não especificado nos decisores, BADAL determina que a cláusula OTHERWISE, obrigatória em todas as conexões, garanta que sempre haverá algum destino para o fluxo do programa após o término da execução de um bloco básico.

Uma entrada de bloco básico pode receber mais de uma conexão, conforme exemplo da Figura 3. Por outro lado, cada valor de saída de um bloco básico deve estar associado a uma única conexão, garantindo que o próximo bloco a executar seja obtido deterministicamente.

Até aqui foi definido o dispositivo não-adaptativo subjacente. A camada adaptativa é introduzida entre a camada de decisores e de conectores. Ela se responsabiliza pela capacidade de alteração do programa em tempo de execução, correspondendo a funções adaptativas. Suas chamadas ficam atreladas às conexões condicionais estabelecidas entre os blocos básicos. A Figura 4 apresenta a arquitetura atualizada com a camada adaptativa.

Na nova camada 3 aparecem todas as funções adaptativas cuja utilização esteja prevista na lógica do programa, e essas são associadas aos conectores da camada 4.

As funções adaptativas devem ser especificadas em algum lugar no corpo do programa adaptativo. A declaração de uma função adaptativa resume-se a indicar as ações de modificação do programa adaptativo, a serem efetuadas em tempo de execução nas ocasiões em que a função for ativada.

BADAL determina que as funções adaptativas se restrinjam a executar ações de inserção ou de remoção, tanto de blocos básicos como de conexões entre eles [6]. As partes do metamodelo que formalizam as funções adaptativas devem refletir esse aspecto.

A referência a um bloco básico deve ser feita por nome, enquanto a referência a uma conexão deve especificar o respectivo bloco básico de origem, bem como o valor de saída desse bloco básico que o seleciona [6].

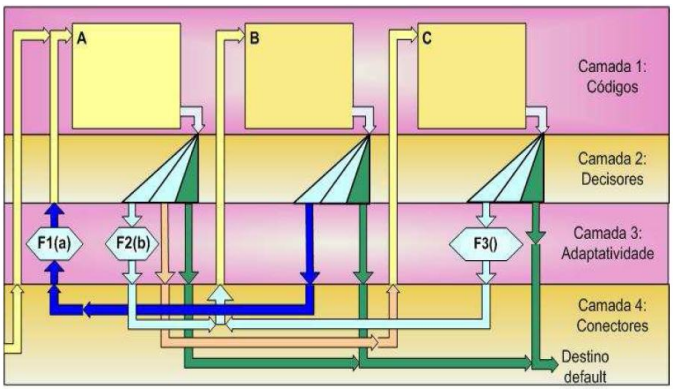

Fig. 4. Arquitetura de um programa adaptativo [6].

A seguinte seção introduzirá o SBMM e apresentará um metamodelo formal para a representação gráfica aqui descrita.

IV. Set Based Meta Modeling (SBMM)

Com a motivação de que o MOF apresenta uma definição com certo grau de complexidade [8], tais como ser descrito através dele mesmo em sua especificação *standalone* e prover capacidades a princípio desnecessárias na camada M3, aliado ao fato de quinze anos após sua publicação ainda não ser adotado em larga escala, pesquisou-se uma alternativa mais simples que deu origem ao SBMM. Trata-se de uma alternativa ao MOF para a camada M3, que permite definir metamodelos do nível M2 através da instanciação de conjuntos e relações pré-definidas bastante simples. Teve também inspiração em ideias de outros trabalhos [9][10].

Um metamodelo nomeado consiste em um nome (cadeia de símbolos) n, um conjunto C de metaclasses nomeadas, um conjunto Γ de generalizações e um conjunto E de enumerações nomeadas. Ou seja, um metamodelo é uma quádrupla MM:

$$MM = (n, C, \Gamma, E) \quad (1)$$

Onde:

- $n \in \Sigma^+$, onde Σ é um conjunto de símbolos pré-definido (fora da definição de MM) para a formação de cadeias que representam nomes. Por exemplo, Σ pode ser o conjunto das letras alfabeto romano com maiúsculas e minúsculas, dígitos de 0 a 9 e símbolos adicionais como *underscore*. O elemento n representa o nome do metamodelo, servindo para implementar capacidade equivalente a dos identificadores do MOF.
- $C = \{c_1, ..., c_n\}$ é o conjunto finito, eventualmente vazio, de metaclasses. Uma metaclasse representa uma abstração de um conceito na camada M2, de forma análoga ao que representa uma classe em uma linguagem de programação orientada a objetos na camada M1.

• $\Gamma \subset C \times C$ é uma relação transitiva livre de ciclos que mapeia submetaclasses em suas supermetaclasses, representando o conceito de herança da orientação a objetos. É transitiva, pois se c_1 é subclasse de c_2 e c_2 é subclasse de c_3, então c_1 é subclasse de c_3. Não é uma relação reflexiva, pois uma metaclasse não é submetaclasse nem supermetaclasse dela mesma. Por não permitir ciclos e nem reflexão, então Γ é subconjunto próprio de $C \times C$.

• $E = \{e_1, ..., e_m\}$é o conjunto finito, eventualmente vazio, de enumerações.

Um metamodelo, e portanto suas metaclasses, existem na camada M2. As definições sobre metaclasses, generalizações e enumerações existem na camada M3. Uma instância de uma metaclasse é chamada de elemento de modelo, e esses existem no nível M1, onde trabalha o desenvolvedor de software.

A metaclasse a partir da qual um elemento de modelo foi instanciado é denominada de seu tipo base (*base type*). As supermetaclasses dessa metaclasse, e também ela mesma, são denominadas de tipos do elemento (*type*). Logo, o tipo base também um tipo do elemento.

Cada metaclasse c_i possui um nome w_i e um conjunto de propriedades P_i. Sendo assim, podemos defini-las como pares ordenados:

$$c_i = (w_i, P_i) \quad (2)$$

Onde:

• $w_i \in \Sigma^+$ é a cadeia de símbolos que nomeia c_i.

• $P_i = \{p_{i1}, ..., p_{in}\}$ é o conjunto finito de propriedades, eventualmente vazio.

Os nomes devem ser únicos dentro do metamodelo, tais que se $c_i \neq c_j$ então $w_i \neq w_j$.

Do ponto de vista semântico, as propriedades $p_{ij} \in P_i$ definem *slots* de informação para elementos de modelos (instâncias) da metaclasse C_i e de suas submetaclasses.

Cada propriedade p_{ij} consiste em um nome, multiplicidade, tipo alvo (que pode ser uma metaclasse ou enumeração) e uma multiplicidade que restringe quantos elementos o *slot* de informação representado consegue armazenar. Isto é:

$$p_{ij} = (v_{ij}, t_{ij}, m_{ij}) \quad (3)$$

Onde:

• $v_{ij} \in \Sigma^+$ é a cadeia de símbolos que nomeia p_{ij}.

• $t_{ij} \in C \cup E$ é o tipo alvo da propriedade, ou seja, pode ser uma metaclasse ou uma enumeração do metamodelo.

• $m_{ij} \in \mathbf{N} \times (\mathbf{N}^+ \cup \{*\})$ é a multiplicidade da propriedade, sendo um par ordenado cujo primeiro elemento é um número natural que denota o limite inferior de *slots* que a propriedade é capaz de armazenar. O segundo elemento pode ser um número natural positivo ou um símbolo *, que representa infinito, denotando o limite superior de *slots*. Se $m_{ij} = (1,1)$, por exemplo, isso significa que a propriedade possui um e apenas um *slot* de informação. Se $m_{ij} = (0,*)$, então a propriedade representa uma lista com um número arbitrário de *slots*. Na UML e no MOF as multiplicidades não necessariamente estão dentro de uma faixa de valores inicial e final, podendo ser valores arbitrários tais como 1, 3 e 5 (sem incluir 2 e 4). Mas para nossos propósitos a definição apresentada aqui é suficiente. Para tornar a notação mais semelhante a da UML e MOF, um par ordenados de multiplicidade (x,y) também será denotado por $x..y$.

Os nomes das propriedades devem ser únicos dentro da metaclasse, tais que se $p_{ij} \neq p_{ik}$ então $v_{ij} \neq v_{ik}$.

As enumerações $e_i \in E$ servem para definir tipos de dados básicos cujas instâncias podem assumir um valor de um conjunto finito, definido pela própria enumeração. Ou seja, cada enumeração pode ser definida por:

$$e_i = (u_i, L_i) \quad (4)$$

Onde:

• $u_i \in \Sigma^+$ é a cadeia de símbolos que nomeia e_i.

• L_i é o conjunto finito de valores que as instâncias de e_i podem assumir.

Os nomes devem ser únicos dentro do metamodelo, tais que se $e_i \neq e_j$ então $u_i \neq u_j$.

Por exemplo, podemos definir uma enumeração de um tipo básico booleano como:

$$e_1 = (\text{“}Boolean\text{”}, \{false, true\}) \quad (5)$$

As cadeias de símbolos que representam nomes são denotadas entre aspas para evitar confusão com referências a outros elementos do metamodelo ou valores permitidos em enumerações, ou seja, não confundir a propriedade p com o eventual nome (cadeia de símbolos) de um elemento do metamodelo “p”.

Estender a definição de enumeração para aceitar conjuntos L_i infinitos é simples, mas não faz sentido na prática pois uma variável de computador é sempre armazenada em um número finito de bits, e portanto uma instância de enumeração na prática não pode assumir um dentre infinitos valores. Sendo assim, um tipo básico de número inteiro de 64 bits com sinal pode ser definido pela seguinte enumeração:

$$e_2 = (\text{“}Integer64\text{”}, \{x \mid (x \in \mathbf{Z}) \wedge (-2^{63} \leq x \leq 2^{63} - 1)\}) \quad (6)$$

Uma enumeração para o tipo String sobre um alfabeto Σ pode ser definida por:

$$e_3 = (\text{“}String\text{”}, \{x \mid (x \in \Sigma^*) \wedge (|x| \leq h)\}) \quad (7)$$

A restrição $|x| \leq h$ para algum limitante superior h garante que a lista seja finita. Na prática, as linguagens de programação permitem variáveis do tipo String de tamanhos bastante elevados, de forma que h é muito grande. Normalmente é limitado pela memória disponível ou alguma característica da linguagem de programação utilizada.

Entretanto, como o SBMM é independente de implementação e usa apenas teorias de conjuntos e de linguagens, para facilitar definições poderão ser utilizadas enumerações com infinitos valores permitidos, tais como:

$$e_4 = (\textit{“Integer”}, \boldsymbol{Z}) \quad (8)$$

Isso porque, do ponto de vista teórico, não há impedimentos para que uma enumeração permita infinitos valores possíveis. Até mesmo conjuntos infinitos não enumeráveis poderiam ser usados, como por exemplo:

$$e_5 = (\textit{“Real”}, \boldsymbol{R}) \quad (9)$$

O leitor deve subentender que, ao ser transportado para uma implementação computacional, os eventuais conjuntos infinitos de valores de enumerações deverão ser substituídos por equivalentes práticos. Por exemplo, a enumeração e_2 apresentada acima é uma implementação usual de e_4, já que inteiros de 64 bits resolvem a maioria dos problemas que precisam manipular inteiros em geral.

As enumerações não contêm propriedades e nem possuem uma relação de generalização.

A. Metamodelo em SBMM para Programas Adaptativos

Para facilitar o entendimento do metamodelo proposto, as sentenças que o compõem serão apresentadas ao longo das explicações.

Seja MM_{PA} um metamodelo que descreve a representação de programas adaptativos apresentada na seção III.

$$MM_{PA} = (n, C, \Gamma, E) \quad (10)$$

Inicialmente define-se n = “AdaptiveProgramMetamodel” como o nome do metamodelo. Introduz-se então no conjunto E duas enumerações de tipos básicos e_1 e e_2, já discutidas anteriormente.

- $e_1 = (\textit{“String”}, \{x \mid (x \in \Sigma^*) \wedge (|x| \leq h)\})$
- $e_2 = (\textit{“Integer”}, \boldsymbol{Z})$

Essas enumerações serão referenciadas nas metaclasses.

Conforme mostrado na Figura 4, identifica-se que um programa adaptativo é composto por quatro componentes fundamentais: bloco básico, decisor, função adaptativa e conexão. Serão criadas metaclasses para abstrair esses conceitos, bem como metaclasses auxiliares.

Seja $c_1 \in C$ a metaclasse que representa um bloco de código básico. Por convenção, todos os nomes de metaclasse serão iniciados com o prefixo MC.

- $c_1 = (\textit{“MCBasicBlock”}, P_1)$
  - $P_1 = \{p_{11}, p_{12}\}$
    - $p_{11} = (\textit{“Name”}, e_1, 1..1)$
    - $p_{12} = (\textit{“Code”}, e_1, 1..1)$

Um bloco básico é composto por um nome e um código-fonte na linguagem hospedeira, ambos representados como as propriedades do tipo String p_{11} e p_{12}. Neste trabalho não se entrará no mérito das regras sintáticas do código na linguagem hospedeira, supondo que este seja um problema à parte. O foco será dado aos aspectos arquiteturais do modelo do programa adaptativo apresentados na Figura 4.

Seja $c_2 \in C$ a metaclasse que abstrai a conexão adaptativa, que conecta a saída de um bloco básico a um ou mais blocos básicos conforme valores de saída.

- $c_2 = (\textit{“MCAdaptiveConnection”}, P_2)$
  - $P_2 = \{p_{21}, p_{22}, p_{23}\}$
    - $p_{21} = (\textit{“From”}, c_1, 1..1)$
    - $p_{22} = (\textit{“ConditionalConnections”}, c_3, 0..*)$
    - $p_{23} = (\textit{“OtherwiseConnection”}, c_4, 1..1)$

A propriedade p_{21} representa o bloco básico de origem da conexão, enquanto p_{22} referencia as possíveis múltiplas conexões condicionais que partem do bloco. Todo bloco deve ter uma e apenas uma conexão *default* caso a saída em tempo de execução fornecida pelo bloco não corresponda a nenhuma conexão condicional. Essa única conexão é representada por p_{23}.

As propriedades p_{22} e p_{23} são *slots* de informação para instâncias de c_3 e c_4 respectivamente. Essas, por sua vez, são metaclasses que representam uma conexão condicional e uma conexão *default*, na ordem. Uma conexão condicional nada mais é que uma conexão *default* acrescida de um valor que define que ela será acionada quando a saída do bloco for igual a este valor. Como possuem aspectos comuns, pode-se definir c_4 inicialmente e depois criar c_3 como submetaclasse de c_4.

- $c_4 = (\textit{“MCGeneralConnnection”}, P_4)$
  - $P_4 = \{p_{41}, p_{42}, p_{43}\}$
    - $p_{41} = (\textit{“To”}, c_1, 1..1)$
    - $p_{42} = (\textit{“AdaptiveFuncCallBefore”}, c_5, 0..1)$
    - $p_{43} = (\textit{“AdaptiveFuncCallAfter”}, c_5, 0..1)$
- $c_3 = (\textit{“MCConditionalConnection”}, P_3)$
  - $P_3 = \{p_{31}\}$
    - $p_{31} = (\textit{“OutputValue”}, e_2, 1..1)$

Para que a metaclasse da conexão condicional c_3 seja de fato submetaclasse de c_4, é necessário introduzir o par ordenado (c_3, c_4) em Γ. Até aqui, portanto, $\Gamma = \{(c_3, c_4)\}$. A semântica do SBMM estabelece que c_3 herda as propriedades de c_4.

Observar que a propriedade p_{31}, que representa o valor de saída que decide a utilização da conexão é definido como tipo inteiro (e_2), de acordo com as definições de [6].

Aparece nas definições de p_{42} e p_{43} a metaclasse c_5, ainda não mencionada. Essa metaclasse deve ser definida de modo a representar uma chamada de função adaptativa. Este tipo de chamada se caracteriza por uma função adaptativa alvo e uma

lista ordenada, eventualmente vazia, de blocos básicos passados como parâmetros, conforme define [6]. Para refletir esses aspectos, cria-se então a metaclasse $c_5 \in C$.

- c_5 = (*"MCAdaptiveFunctionCall"*, P_5)
  - P_5 = {p_{51}, p_{52}}
    - p_{51} = (*"AdaptiveFunction"*, c_7, 1..1)
    - p_{52} = (*"ParametersValues"*, c_6, 0..*)

Uma vez que a multiplicidade da propriedade p_{52} é 0..*, o que significa que a chamada pode passar um número arbitrário de parâmetros (inclusive nenhum), é necessário introduzir a metaclasse c_6 que representa um par nome/valor, evitando não determinismos caso haja mais de um parâmetro sendo passado. Os nomes dos parâmetros da chamada devem corresponder aos nomes dos parâmetros de entrada definidos na especificação da função adaptativa.

- c_6 = (*"MCAdaptiveFuncParamValue"*, P_6)
  - P_6 = {p_{61}, p_{62}}
    - p_{61} = ("ParameterName", e_1, 1..1)
    - p_{62} = ("ParameterValue", c_1, 1..1)

Continuando a criação do metamodelo, estabelece-se a metaclasse c_7 para representar as funções adaptativas em si, lembrando que c_5 representa apenas a chamada de uma função adaptativa associada a um conector, e não a especificação da função em si. A metaclasse c_7 fará esse papel.

- c_7 = (*"MCAdaptiveFunction"*, P_7)
  - P_7 = {p_{71}, p_{72}, p_{73}}
    - p_{71} = (*"Name"*, e_1, 1..1)
    - p_{72} = (*"ParametersNames"*, e_1, 0..*)
    - p_{73} = (*"Code"*, e_1, 1..1)

Assim como este metamodelo não entrou no mérito do código dos blocos básicos em linguagem hospedeira, representado por p_{12}, tendo sido o mesmo modelado apenas como uma String, o mesmo se aplica a p_{73}, servindo como *slot* para armazenar o código que implementa a função adaptativa em linguagem de programação adaptativa.

Por definição, os parâmetros das funções adaptativas correspondem a blocos básicos, não podendo ser de outro tipo. Por isso não é necessário prever propriedades para tipos de parâmetros.

Por fim, cria-se a metaclasse c_0, que estabelece os blocos de entrada e saída do programa adaptativo, conforme especificado em [6]. Serve como agregador principal dos blocos, conexões e funções adaptativas.

- c_0 = (*"MCAdaptiveProgram"*, P_0)
  - P_0 = {p_{01}, p_{02}, p_{03}, p_{04}, p_{05}, p_{06}}
    - p_{01} = (*"Name"*, e_1, 1..1)
    - p_{02} = (*"EntryBlock"*, c_1, 1..1)
    - p_{03} = (*"ExitBlock"*, c_1, 1..1)
    - p_{04} = (*"OtherBlocks"*, c_1, 0..*)
    - p_{05} = (*"AdaptiveConnections,* c_2, 1..*)
    - p_{06} = (*"AdaptiveFunctions"*, c_7, 0..*)

Como restrição, um modelo de programa adaptativo construído sobre o metamodelo aqui proposto deve possuir uma e apenas uma instância de c_0. Ou seja, não é permitido que um modelo contenha mais de um programa adaptativo, e nem que haja ausência do mesmo.

Um aspecto interessante é que neste metamodelo as conexões que saem dos blocos foram definidas não como propriedades dos próprios blocos (metaclassse c_1), mas sim como elementos externos (metaclasse c_2). Esta forma é coerente com o fato de enxergar a camada de conectores e de adaptatividade da Figura 4 como desacopladas das definições dos blocos, podendo ser remanejadas em cada modelo de programa adaptativo enquanto se reusa blocos já existentes de outros programas prévios. Como restrição, em um modelo de programa adaptativo não pode haver mais de uma instância da metaclasse c_2 que referencia o mesmo bloco na propriedade p_{21}, do contrário pode ocorrer não determinismos em tempo de execução.

Em resumo, os conjuntos principais do metamodelo MM_{PA} são:

- C = {c_0, c_1, c_2, c_3, c_4, c_5, c_7}
- Γ = {(c_3,c_4)}
- E = {e_1, e_2}

O SBMM propõe uma notação gráfica similar a do MOF e UML, onde se representa as metaclasses e enumerações por retângulos nomeados. As propriedades são representadas por linhas que ligam a metaclasse ao tipo alvo, que pode ser também uma metaclasse ou enumeração. As linhas são rotuladas pelo nome da propriedade e direcionadas com uma seta ao tipo alvo. Nesta notação, o metamodelo MM_{PA} está representado pela Figura 5.

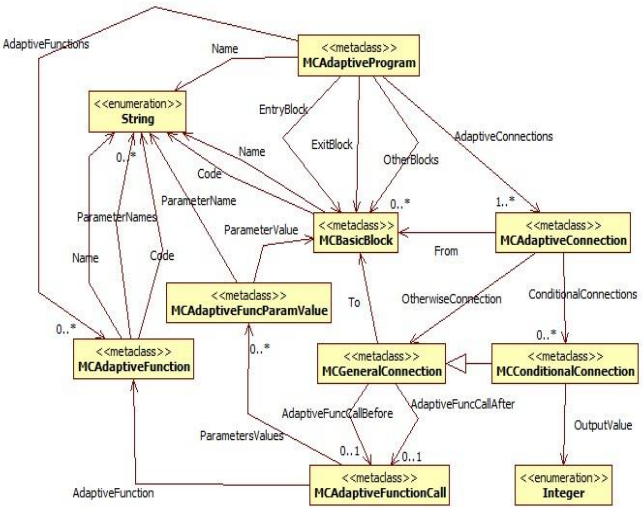

Fig. 5. Notação gráfica do metamodelo.

V. Resultados e Discussão

A seção anterior apresentou o resultado da descrição de um metamodelo para uma representação de programas adaptativos utilizando SBMM. A descrição resultante é puramente conceitual, totalmente independente de qualquer tipo de implementação. Além disso, provê um formalismo que captura os elementos da essência da definição de programas adaptativos. É extensível, ou seja, se forem utilizadas extensões na definição de programa adaptativo, o metamodelo pode ser ajustado através da substituição, remoção ou inserção de elementos que reflitam formalmente a definição modificada.

Um dos trabalhos de pesquisa atuais é o desenvolvimento da SBMM Tool (SBMMT), ferramenta capaz de prover edição de modelos genéricos de acordo com metamodelos em SBMM. O presente trabalho viabiliza a utilização futura da SBMMT para edição de modelos de programas adaptativos, podendo em um próximo passo prover geração automática de código BADAL. Também fica encaminhada a potencial geração de código em outras futuras eventuais linguagens de programação adaptativas a partir dos mesmos modelos de origem. No entanto, um PIM descrito no metamodelo proposto será capaz de gerar código adaptativo em uma ou mais linguagens apenas no que diz respeito à estrutura dos blocos e conexões. A implementação dos blocos e funções adaptativas em si foram modeladas como propriedades String, ou seja, a SBMMT ou outra ferramenta que permita o programador editar um modelo na forma da representação gráfica do programa adaptativo considera que essas implementações são texto livre, devendo o programador preencher código na linguagem hospedeira e linguagem adaptativa de interesse. Futuras extensões do metamodelo proposto podem considerar detalhes dos aspectos de implementação ao menos das funções adaptativas, permitindo que o PIM contenha informações completas sobre sua implementação sem depender de nenhuma linguagem específica.

VI. Conclusão

O presente trabalho mostra a viabilidade e caminhos para utilizar uma técnica muito poderosa, a MDA, para a criação de programas adaptativos. A MDA é considerada por alguns a grande tendência futura da engenharia de software, embora atualmente pouco explorada na prática [2]. Programas adaptativos representam uma nova abordagem para resolver problemas. O caminho apontado pelo trabalho é relevante na medida em que traz os ganhos propostos pela MDA para a área. Isso ocorreria por meio de ferramentas CASE para programas adaptativos com possibilidade de gerar código, nas quais os modelos serviriam não só como documentação, mas também como artefatos de construção dos programas adaptativos. Na situação ideal, funções de mapeamento podem ser criadas para gerar o programa para distintas plataformas ou linguagens a partir de um mesmo modelo. A possibilidade da utilização conjunta de ambas as técnicas é, portanto, de interesse não só para a engenharia de software como também para os campos onde programas adaptativos podem ser aplicados.

VII. Referências

[1] OMG, MDA Guide Version 1.0.1. Disponível em: http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[2] Ambler, S., The Object Primer: Agile Modeling-Driven Development with UML 2.0. New York: Cambridge University Press, 2004.

[3] OMG, Model Driven Architecture Executive Overview. Disponível em: http://www.omg.org/mda/ executive_overview.htm.

[4] OMG, Model Driven Architecture FAQ. Disponível em: http://www.omg.org/ mda/faq_mda.htm.

[5] Object Management Group web site. Disponível em: http://www.omg.org/

[6] Silva, S. R. B., Software Adaptativo: Método de Projeto, Representação Gráfica e Implementação de Linguagem de Programação. Dissertação (Mestrado). Escola Politécnica da Universidade de São Paulo, 2010.

[7] OMG, Meta Object Facility (MOF) Core Specification. Disponível em: http://www.omg.org/spec/MOF/2.4.1

[8] Bézivin, J., Gerbé, O., Towards a precise definition of the OMG/MDA framework. In: Automated Software Engineering, 2001. Proceedings. 16th Annual International Conference on (pp. 273-380). IEEE.

[9] Favre, J. M., Towards a basic theory to model Model Driven Engineering. In: 3rd Workshop in Software Model Engineering, WiSME, 2004.

[10] Alanen, M., Porres, I., A Relation Between Context-Free Grammars and Meta Object Facility Metamodels, Technical report, Turku Centre for Computer Science, 2003.

[11] Neto, J. J., *Adaptive Rule-Driven Devices - General Formulation and Case Study*. Lecture Notes in Computer Science. Watson, B.W. and Wood, D. (Eds.): Implementation and Application of Automata 6th International Conference, CIAA 2001, Vol. 2494, Pretoria, South Africa, July 23-25, Springer-Verlag, 2001, pp. 234-250.

[12] Hirakawa, A. R., Saraiva, A. M., Cugnasca, C. E., *Adaptive Automata Applied on Automation and Robotics (A4R)*. Latin America Transactions, IEEE, 2007. 5(7), 539-543.

[13] Pelegrini, E. J., Códigos Adaptativos e Linguagem para Programação Adaptativa: Conceitos e Tecnologia. Dissertação (Mestrado). Escola Politécnica da Universidade de São Paulo, 2009.

[14] De Lara, J., Vangheluwe, H., Using AToM as a Meta-CASE Tool. In: Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS), 2002.

[15] Mellor, S., Scott, K., Uhl, A., e Weise, D., MDA Distilled: Principles of Model-Driven Architecture. Boston: Pearson Education Inc., 2004.

Sergio R. M. Canovas nasceu em São Paulo, Brasil, em 1981. Graduou-se e recebeu o título de mestre em Engenharia Elétrica pela Universidade de São Paulo (USP), São Paulo, Brasil em 2003 e 2006, respectivamente. Em 2011 ingressou no programa de doutoramento em Engenharia Elétrica pela mesma universidade, sendo pesquisador no Laboratório de Automação Agrícola (LAA). Seus interesses de pesquisa incluem redes de controle, sistemas ERP, MDE/MDA, formalismos para metamodelagem e transformação de modelos de software.

Carlos E. Cugnasca graduou-se em Engenharia Elétrica na Escola Politécnica da Universidade de São Paulo (EPUSP), Brasil, em 1980. Na mesma instituição, obteve o seu mestrado (1988), doutorado (1993) e Livre-Docência (2002). Suas principais atividades de pesquisas incluem instrumentação inteligente, automação agrícola e agricultura de precisão. É professor do Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP desde 1988, ocupando atualmente a função de Professor Associado. Vem desenvolvendo pesquisas sobre redes de controle, redes de sensores sem fio, eletrônica embarcada e aplicações de computação pervasiva.