Um Mapeamento de Modelos Adaptativos para Dispositivos Adaptativos Guiados por Regras

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

***Abstract* — *Model Driven Engineering* (MDE) é uma abordagem para desenvolvimento de software em que modelos são artefatos fundamentais a serem construídos e manipulados, podendo gerar código-fonte automaticamente. Uma das possibilidades é a execução direta de modelos, e para isso caracterizam-se linguagens de modelagem interpretáveis. Recentemente, surgiu o conceito de modelos adaptativos, que são modelos executáveis com capacidade de passar por alterações estruturais durante sua execução através de autorresposta a estímulos externos. Este trabalho apresenta um mapeamento de tais modelos para uma formulação geral de dispositivos adaptativos guiados por regras encontrada na literatura. Como estratégia, um padrão de projeto de linguagens de modelagem executáveis é utilizado e estendido para incluir conceitos de adaptatividade. Algumas outras restrições são adicionadas aos modelos para que se possa identificar de forma geral todos os elementos necessários para mapeá-los. Com isso, modelos adaptativos são colocados no contexto geral da área de adaptatividade, aproveitando-se as teorias subjacentes já existentes.**

***Keywords*— Tecnologias adaptativas (*adaptive Technologies*), modelos executáveis (*executable models*) modelos adaptativos (*adaptive models*), *Model Driven Engineering*, *Set Based Meta Modeling*.**

I. INTRODUÇÃO

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 definem seu comportamento [3]. 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 das regras de comportamento do dispositivo não-adaptativo subjacente ou também do próprio mecanismo adaptativo.

Um autômato adaptativo, por exemplo, é um dispositivo adaptativo que estende o conceito de autômato finito incorporando a característica de desenvolver uma autorreconfiguração em resposta a um estímulo externo [4]. Criação de novos estados e transições em tempo de execução são exemplos de autorreconfiguração.

A tecnologia adaptativa traz novos mecanismos a serem utilizados em diferentes abordagens para resolver problemas, podendo ser aplicada em diversas áreas tais como automação e robótica [4], reconhecedores gramaticais [8] e até mesmo na área agrícola [9].

Um programa adaptativo pode ser entendido como uma especificação de uma sequência automodificável de instruções, que representa um código dinamicamente alterável [2]. Podem ser considerados dispositivos adaptativos em que o dispositivo não-adaptativo subjacente seria um programa estático, ou seja, um programa tradicional que não tem sua estrutura modificada ao longo de sua execução.

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), por exemplo, é uma linguagem de programação adaptativa de alto nível proposta por [2]. 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 [2] gera código para o ambiente de execução desenvolvido em [5], 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 [2], descrita em linguagem natural. Esta representação é formalizada em [7][10] com a utilização do *Set Based Meta Modeling* (SBMM) [20], um formalismo de metamodelagem. Técnicas para codificar programas adaptativos utilizando linguagens de programação orientadas a objetos usuais são apresentadas por [16].

Model Driven Engineering (MDE), um outro tópico de pesquisa em Engenharia de Software, é uma abordagem para desenvolvimento de software através da qual um sistema é construído pela transformação de modelos em código-fonte em alguma linguagem alvo. Essa transformação pode ocorrer em um número arbitrário de passos. Um modelo de um sistema é uma descrição ou especificação do mesmo considerando um determinado propósito, e, por isso, pode não representar todos os aspectos e propriedades do sistema por si só [6]. Porém, no cenário ideal, o código-fonte do sistema, em um estado pronto para ser compilado, seria completamente gerado a partir de transformações de seus modelos.

Com isso, os modelos de software servem não apenas como documentação, mas também como artefatos fundamentais para a construção do programa. Aumenta-se o nível de abstração no desenvolvimento e ganha-se na possibilidade de geração do mesmo sistema para diversas plataformas, uma vez que o

[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@poli.usp.br).

mesmo modelo poderia dar origem a código-fonte em mais de uma linguagem alvo, desde que se tenham as transformações necessárias disponíveis.

Usualmente um modelo pode ser representado por uma combinação de textos e desenhos, podendo estar descrito em linguagem de modelagem (ex: UML) ou em linguagem natural [1]. Entretanto, para que a MDE seja possível na prática, é necessário que os modelos de software estejam descritos em linguagens que permitam a leitura por programas sendo, portanto, a linguagem natural inadequada para este fim. Este problema é atacado pelos metamodelos.

Metamodelos são modelos de um tipo especial que descrevem a sintaxe abstrata de uma linguagem de modelagem. Existem, por exemplo, metamodelos que especificam a UML. Eles determinam as abstrações e conceitos que podem ser instanciados em seus modelos, bem como as restrições aplicáveis [11]. Em um diagrama de classes UML, como exemplo de restrição, não pode haver uma associação desconectada de classes. É o metamodelo que estabelece o conceito de associação e a restrição de que cada instância deve ter seus fins conectados em classes. O exemplo da Fig. 1 apresenta um metamodelo para máquinas de estado simples [11] e uma máquina de estado específica, que é um modelo conforme ao metamodelo. O metamodelo está denotado na linguagem *Meta Object Facility* (MOF). Restrições costumam ser definidas em alguma linguagem textual, tal como *Object Constraint Language* (OCL), e não são expressas no diagrama do metamodelo.

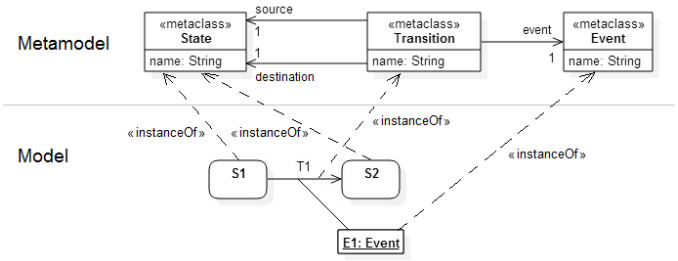

Fig. 1. Exemplo de um metamodelo para máquinas de estado e uma máquina de estado específica. Fonte: [11] (adaptado).

Alguns modelos de software possuem caráter descritivo, tais como diagramas de classes ou entidade-relacionamento, enquanto outros possuem caráter executável, tais como máquinas de estado ou redes de Petri. Linguagens de modelagem específicas de domínio, ou *domain specific modeling languages* (DSMLs) que permitem a criação de modelos executáveis são conhecidas como *executable DSMLs* (x-DSMLs). A execução de modelos conformes a essas linguagens pode ser alcançada não apenas por transformação de modelos em código-fonte, mas também por interpretação direta dos modelos. Linguagens com esta capacidade são denotadas pela sigla i-DSML e seus modelos são interpretados de acordo com uma semântica operacional processada por um motor (*engine*) de execução. Com esta abordagem, nenhuma transformação ocorre e o modelo é diretamente executável [12].

Como modelos executáveis são representações de programas de computador em mais alto nível de abstração, e alguns deles podem ser executados diretamente, então faz sentido considerar modelos adaptativos, uma vez que existem programas adaptativos. Modelos executáveis são estudados em diversos trabalhos tais como [13][14][15], entre outros. Porém, até onde nosso conhecimento permite afirmar, a adaptatividade de modelos executáveis parece ter sido estudada primeiramente no contexto da MDE por [12] e [18], sendo também abordada por [19], consistindo ainda um campo de pesquisa a ser explorado.

Este trabalho tem por objetivo utilizar as ideias apresentadas por [12] sobre caracterização de modelos adaptativos e o padrão de projeto de i-DSMLs apresentado por [15] para estabelecer um mapeamento de modelos adaptativos para a formulação geral de dispositivos adaptativos proposta por [3]. Assim, caracteriza-se esse tipo de modelo dentro dessa categoria de dispositivos, e permite-se aproveitar resultados existentes através dessa conexão. Ademais, teorias e ferramentas para metamodelagem têm evoluído de forma independente de modelos de computação, criando um *gap* cultural e técnico entre essas duas comunidades [17]. Busca-se com este trabalho contribuir com alguma redução neste *gap*.

A seção II apresenta a conceituação de modelos executáveis e modelos adaptativos encontrada na literatura. A seção III apresenta uma revisão da formulação geral para dispositivos adaptativos guiados por regras, enquanto a seção IV apresenta o mapeamento que é objetivo deste trabalho. A seção V discute os resultados e é seguida pela seção VI, que apresenta as conclusões. Por fim, a seção VII lista as referências bibliográficas.

II. Modelos Executáveis e Modelos Adaptativos

Esta seção apresenta a conceituação de modelos executáveis e modelos adaptativos encontrada na literatura. Ela é usada como base para o mapeamento apresentado mais adiante na seção IV.

A. Modelos Executáveis

Um modelo executável é um tipo especial de modelo que pode ser executado. Sua definição é discutida em diversos *papers* tais como [13][14][15], tendo sido estabelecido o seguinte consenso [12]:

- Alguns tipos de modelos são executáveis, enquanto outros não;
- Um motor (*engine*) é responsável por executar o modelo, tomando-o como entrada e efetuando o processamento necessário;
- Se o modelo armazena as informações necessárias para controle de sua própria execução através de um motor, então ele é dito autocontido (*self-contained*).

Um exemplo de modelo autocontido para máquinas de estado seria se o metamodelo previsse uma propriedade booleana *isCurrent* para a metaclasse que representa o estado e uma restrição determinando que só pode haver um estado corrente por vez. Em um modelo em execução, essa propriedade carregaria o valor *True* no estado corrente, dentro do modelo, e *False* para os outros. Se o gerenciamento do

estado corrente ocorre exclusivamente no motor, estando essa informação fora do modelo, então o modelo não é autocontido.

A estratégia dos modelos autocontidos parece poluir o metamodelo com detalhes não relevantes em tempo de projeto, mas eles podem ser justificados pelo fato de que a executabilidade é parte da natureza do modelo e, portanto, este pode ter seu nível de detalhes aumentado para atender este requisito [12]. Outra razão é que essa estratégia oferece a vantagem de que o estado corrente de execução pode ser serializado em um arquivo de saída, provendo um mecanismo de rastreabilidade da execução como sequências de modelos. Isso possibilitaria a oferta de operações úteis tais como *rollbacks*, depuração e teste.

Existe uma classificação geral que pode ajudar a identificar modelos executáveis [12]: a classificação produto ou processo. Modelos podem expressar produtos ou processos, independentemente do sistema sob consideração. O ponto central é que modelos de processos habilitam a executabilidade de seu conteúdo pois contêm conceitos relacionados a execução: ponto de início, ponto de fim, passo de evolução, etc. Por outro lado, modelos de produto são mais estruturais, tais como diagramas de classes UML, e eles não expressam diretamente comportamento ou interações.

Não consideramos essa classificação muito precisa para determinar modelos executáveis. Observemos um exemplo de modelo que descreve uma interface de usuário do tipo *Create/Retrieve/Update/Delete* (CRUD) para objetos persistentes em aplicativos de negócio [11]. O metamodelo correspondente define uma DSML que prevê apenas elementos estruturais da interface de usuário, podendo seus modelos serem classificados como modelos de produto. Entretanto, não podemos afirmar que esses modelos não são executáveis. Um motor, que implementa a semântica de execução, pode ser capaz de renderizar a interface e processar todas as interações com o usuário, tais como o clique em um botão para inserir um objeto. Em todo o caso, esta classificação ainda provê alguma evidência sobre a executabilidade de um dado tipo de modelo.

Para dar continuidade no mapeamento de modelos adaptativos, precisamos considerar apenas os modelos executáveis de processo. Sendo assim, deve-se pressupor que um modelo de produto executável pode ser mapeado em um modelo de processo equivalente. Sempre pode ser criado um modelo de processo de mais baixo nível de abstração (em outra linguagem de modelagem) que incorpore explicitamente conceitos implícitos no modelo de mais alto nível. Mesmo que esse mapeamento não seja de interesse da MDE, pois esta abordagem clama pelo aumento do nível de abstração, é de interesse para a caracterização e mapeamento desse tipo de modelo. Este princípio pode ser argumentado pela analogia com programas de computador: um programa escrito em linguagem de alto nível, que torna implícitos detalhes computacionais de execução, sempre pode ser reescrito ou compilado em um código equivalente em linguagem de mais baixo nível, no qual detalhes computacionais de execução são mais evidentes.

Um padrão recorrente pode ser observado nos constituintes da i-DSML [12][15]:

- Parte estática ou estrutural: metaelementos que expressam a organização estrutural do processo modelado;
- Parte dinâmica: metaelementos que expressam o estado do modelo sendo executado;
- Semântica de execução: expressa como o modelo se comporta enquanto estiver sendo executado.

Os metaelementos estruturais da parte estática são implementados por metamodelagem tradicional. No metamodelo da máquina de estados, eles seriam as metaclasses que representam estados, transições e eventos, por exemplo. Os metaelementos de estado da parte dinâmica têm por objetivo armazenar o estado do modelo em um dado ponto de execução no tempo. São eles que tornam os modelos autocontidos. Essas duas partes compõem, parcialmente, o metamodelo da i-DSML.

Além de considerar modelos de processo, também precisaremos considerar apenas modelos autocontidos. Se o metamodelo da i-DSML sob consideração não previr esses metaelementos, deve-se completá-lo para tal.

A semântica de execução pode ser definida de várias maneiras. Uma semântica axiomática permite completar a especificação do metamodelo com regras sobre como o modelo evolui em tempo de execução. Uma semântica translacional consiste em defini-la através de mapeamento com a semântica de execução de outro dispositivo executável já conhecida. Uma semântica operacional, usada mais adiante neste trabalho, consiste em definir o comportamento de execução em termos de ações através da implementação de um motor capaz de interpretar os modelos [12].

B. Modelos Adaptativos

Um modelo adaptativo é um modelo executável com capacidade de automodificação ao longo de sua execução. Uma i-DSML adaptativa permite a elaboração de modelos adaptativos e esta estende o conceito de i-DSML através da adição de duas partes [12]:

- Parte adaptativa: metaelementos que expressam em que situações deve ocorrer adaptatividade durante a execução do modelo através de ações adaptativas, definidas mais adiante;
- Semântica adaptativa: expressa quais são os efeitos decorrentes das ações adaptativas.

Os metaelementos da parte adaptativa expressam as possibilidades de ocorrência da adaptatividade durante a execução dos modelos.

Um autômato adaptativo, por exemplo, poderia criar nele mesmo novos estados e transições em tempo de execução. A definição de propriedades que permitem especificar ações adaptativas de inserção de estados e transições são exemplos de metaelementos da parte adaptativa, conforme ilustrado pela Fig. 2 [16]. Nesse autômato *M*, a transição de q_1 para ele mesmo dispara uma ação adaptativa definida pela chamada da função adaptativa $\mathscr{F}(p)$ com o argumento $p = q_2$, que irá criar um novo estado entre q_1 e q_2 e modificar as transições para que o caminho entre q_1 e q_2 seja linear e através de símbolos *b*, passando pelos estados intermediários criados. Tal autômato é

capaz de reconhecer a linguagem $L(M) = \{a^n b^n, n \in \mathbf{N}, n \geq 1\}$, e a definição completa de $\mathcal{F}$ pode ser encontrada em [16], onde são apresentados outros detalhes necessários.

A semântica adaptativa também pode ser especificada de várias maneiras, tais como axiomática e operacional. O motor de execução adaptativo de determinada i-DSML consiste no motor de execução original não-adaptativo estendido com capacidades adaptativas [12]. Ele é composto por um conjunto de funções adaptativas básicas que são chamadas durante a execução do modelo quando as capacidades adaptativas são acionadas. As chamadas específicas das funções adaptativas, com valores atribuídos aos argumentos, são as ações adaptativas.

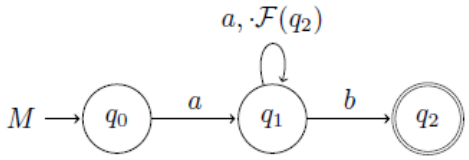

Fig. 2. Autômato adaptativo M que reconhece cadeias na forma $a^n b^n$ com $n \in \mathbf{N}$, $n \geq 1$. Fonte: [16].

Todas as partes da i-DSML adaptativa (estática, dinâmica, adaptativa e semânticas de execução e adaptativas) poderiam, em tese, ser alvo das ações adaptativas. Isso significa que a adaptatividade da adaptatividade (meta-adaptatividade) é possível quando a parte adaptativa está incluída como alvo possível das funções adaptativas.

III. Dispositivos Adaptativos Guiados por Regras

Esta seção apresenta uma revisão da formulação geral para dispositivos adaptativos guiados por regras [3], que são máquinas formais que permitem a alteração de seu comportamento dinamicamente como resposta a estímulos de entrada e sem interferência de agentes externos. A formulação se inicia através da especificação de dispositivos não-adaptativos. Em seguida, é apresentada a formulação de dispositivos adaptativos pela introdução de um mecanismo adaptativo sobre o dispositivo não-adaptativo subjacente. Nosso interesse é, na sequência, mapear modelos executáveis em dispositivos não-adaptativos guiados por regras e estabelecer modelos adaptativos como dispositivos adaptativos em que o dispositivo não-adaptativo subjacente seja o modelo executável não-adaptativo.

Um dispositivo não-adaptativo guiado por regras é todo e qualquer dispositivo formal cujo comportamento depende, de forma exclusiva, de um conjunto finito de regras que definem o mapeamento entre configurações [3]. O dispositivo é dito ser determinístico se, e somente se, para uma dada configuração inicial ou intermediária e um estímulo de entrada, seu conjunto de regras determina uma e apenas uma próxima configuração. Um dispositivo não-adaptativo ND é uma sêxtupla de acordo com (1).

$$ND = (C, NR, S, c_0, A, NA) \quad (1)$$

Em que:

- C é conjunto de todas as possíveis configurações de ND;
- NR é o conjunto de regras que descrevem ND;
- S é o conjunto de todos os possíveis eventos que são estímulos válidos de entrada para ND, com $\varepsilon \in S$, onde ε denota "vazio" e representa o elemento nulo do conjunto;
- $A \subseteq C$ é o subconjunto das configurações de aceitação;
- NA é o conjunto finito, com $\varepsilon \in NA$, de todos os possíveis símbolos que podem ser saídas de ND como efeitos da aplicação de regras em NR. Esses símbolos de saída podem ser mapeados em chamadas de procedimentos. Cadeias de símbolos de saída, portanto, podem ser interpretadas como sequências de chamadas.

Uma regra $r \in NR$ é uma quádrupla $r = (c_i, s, c_j, z)$ em que:

- $c_i \in C$ é a configuração de origem da regra r;
- $s \in S$ é o evento que dispara a regra r se a configuração corrente do dispositivo for c_i;
- $c_j \in C$ é a configuração de destino da regra r;
- $z \in NA$ é o símbolo a ser adicionado ao fim da cadeia de saída quando a regra r é disparada.

A regra r é dita ser compatível com a configuração corrente $c_{corrente}$ se, e somente se, $c_i = c_{corrente}$ [16]. A aplicação de r move o dispositivo da configuração c_i e este fato é denotado por $c_i \Rightarrow^s c_j$. Adicionalmente, z é adicionado ao fim da cadeia de saída do dispositivo. Notar que s, z ou ambos podem ser vazios.

Faça-se $c_i \Rightarrow^{\sim} c_m$ $(m \geq 0)$ denotar $c_i \Rightarrow^{\varepsilon} c_1 \Rightarrow^{\varepsilon} c_2 \Rightarrow^{\varepsilon} \ldots \Rightarrow^{\varepsilon} c_m$, que é uma sequência opcional de movimentos vazios. Faça-se também $c_i \Rightarrow^{\sim wk} c_j$ denotar $c_i \Rightarrow^{\sim} c_m \Rightarrow^{wk} c_j$, uma sequência opcional de movimentos vazios seguidas por uma não-vazia consumindo o símbolo w_k. Uma cadeia de entrada $w = w_1 w_2 \ldots w_n$ é dita ser aceita por ND quando $c_0 \Rightarrow^{\sim w1} c_1 \Rightarrow^{\sim w2} \ldots \Rightarrow^{\sim wn} c_n \Rightarrow^{\sim} c_{final}$, ou $c_0 \Rightarrow^{\sim w} c_{final}$, com $c_{final} \in A$.

Um dispositivo adaptativo AD é construído sobre um dispositivo não-adaptativo subjacente inicial ND_0 pela introdução de um mecanismo adaptativo. Seja k o passo de operação adaptativa de AD. Ele inicia com o valor zero e é automaticamente incrementado de um quando uma ação adaptativa é executada. Ele não é incrementado quando não ocorre uma ação adaptativa. Nesse caso, diz-se também que ocorreu uma ação adaptativa nula. Para cada passo de operação $k \geq 0$, AD segue o comportamento de ND_k até a próxima execução de uma ação adaptativa não-nula. ND_k denota o dispositivo não-adaptativo subjacente de AD no passo de operação adaptativa k.

Em qualquer $k \geq 0$, ND_k é definido pelo seu conjunto de regras NR_k. A execução de uma ação adaptativa não-nula incrementa k de 1 e faz ND_k evoluir para ND_{k+1}. Assim, AD inicia a operação no estágio $k + 1$ pela criação do conjunto NR_{k+1} como uma versão editada de NR_k. AD passará então a seguir o comportamento de ND_{k+1} até que outra ação adaptativa não-nula cause a evolução para ND_{k+2}. Este procedimento itera até que a cadeia de entrada seja totalmente processada.

O dispositivo adaptativo no estágio k é representado por (2). Essa representação inclui informação sobre o dispositivo não-adaptativo subjacente ND_k, que são C_k, S, c_k, A_k e NA.

$$AD_k = (C_k, AR_k, S, c_k, A_k, NA, BA, AA) \quad (2)$$

Assim, AD inicia sua operação na configuração c_0. A execução de uma ação adaptativa não-nula leva AD_k para $AD_{k+1} = (C_{k+1}, AR_{k+1}, S, c_{k+1}, A_{k+1}, NA, BA, AA)$, em que:

- AR_k é o conjunto de regras adaptativas conforme detalhado abaixo;
- BA e AA são os conjuntos de ações adaptativas "antes" (*before*) e "depois" (*after*), respectivamente, com $\varepsilon \in BA$ e $\varepsilon \in AA$.

O conjunto de regras adaptativas AR_k é dado por uma relação de acordo com (3).

$$AR_k \subseteq BA \times C_k \times S \times C_k \times NA \times AA \quad (3)$$

Isso quer dizer que cada regra adaptativa $ar \in AR_k$ é uma sêxtupla (ba, c_i, s, c_j, z, aa) significando que, em resposta a um estímulo de entrada $s \in S$, ar primeiro executa a ação adaptativa "antes" $ba \in BA$ (a execução de ba é abortada se ela elimina ar de AR_k), aplica a regra não-adaptativa subjacente $r = (c_i, s, c_j, z) \in NR_k$ e finalmente executa a ação adaptativa "depois" $aa \in AA$.

O leitor deve notar que NR_k não está explicitado em (2) pois está implícito em AR_k, descartando o primeiro e último elemento da sêxtupla.

Para que o dispositivo seja determinístico, para cada regra não-adaptativa proveniente de NR_k, existe um único par de ações adaptativas "antes" e "depois". Assim, o conjunto AR_k, relacionado ao estágio adaptativo k, pode ser obtido adicionando os devidos pares de ações adaptativas a cada regra não adaptativa pertencente a NR_k.

As ações adaptativas pertencentes a BA e AA podem ser definidas em termos de abstrações chamadas funções adaptativas, de modo similar às chamadas de funções em linguagens de programação usuais [16]. A especificação de uma função adaptativa inclui os seguintes elementos: (a) um nome simbólico, (b) parâmetros formais que referenciam valores passados como argumentos, (c) variáveis que armazenam valores de uma ação elementar de inspeção, (d) geradores que referenciam valores novos a cada utilização, e (e) o corpo da função.

São definidos três tipos de ações adaptativas elementares que realizam testes nas regras ou modificam regras existentes, a saber:

- Ação adaptativa elementar de inspeção: a ação não modifica o conjunto de regras, mas permite a inspeção deste para a verificação de regras que obedeçam um determinado padrão;
- Ação adaptativa elementar de remoção: a ação remove regras que correspondem a um determinado padrão do conjunto de regras;
- Ação adaptativa elementar de inclusão: a ação insere uma regra que corresponde a um determinado padrão no conjunto corrente de regras.

Essas ações adaptativas elementares podem ser utilizadas no corpo de uma função adaptativa, incluindo padrões de regras que utilizem parâmetros formais, variáveis e geradores disponíveis [3][16].

IV. De Modelos Adaptativos Para Dispositivos Adaptativos Guiados por Regras

Esta seção apresenta um mapeamento de modelos adaptativos para a formulação de dispositivos adaptativos guiados por regras de [3] apresentado na seção III. Primeiramente, um modelo executável é mapeado em um dispositivo não-adaptativo seguindo a forma de (1). Para que isso seja possível de algum modo, recorre-se ao padrão de projeto de i-DSMLs proposto por [15]. Em seguida, o padrão é estendido para incluir uma parte adaptativa, a qual é mapeada no mecanismo adaptativo.

A. Padrão de Projeto para i-DSMLs

Seja L uma i-DSML estabelecida pelo metamodelo MM_L. Assume-se que L seja uma linguagem de modelos de processo autocontidos, conforme discutido na seção II. Ou seja, MM_L contém explicitamente metaelementos que expressam a parte dinâmica dos modelos, a qual engloba representação dos estados de execução possíveis, conforme visto na seção II.

Em uma i-DSML de máquinas de estado, por exemplo, essa parte do metamodelo pode ser expressa por uma propriedade *booleana* *isCurrent* na metaclasse *State* e uma restrição indicando que só pode haver um único estado do modelo (instância de *State*) com o valor *True* para esta propriedade em cada passo de execução. A Fig. 3 apresenta uma versão atualizada da metaclasse *State* da Fig. 1 contendo esta propriedade.

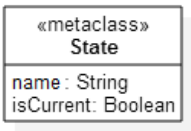

Fig. 3. Metaclasse *State* estendida com propriedade da parte dinâmica.

Em uma i-DSML de redes de Petri, como exemplo adicional, o estado de execução do modelo pode ser expresso por uma propriedade *numberOfTokens* na metaclasse que representa o conceito de posição, e esta armazenará um número inteiro (eventualmente com alguma restrição limitadora) para cada instância de posição nos modelos executados.

Nos exemplos dos dois últimos parágrafos, a parte dinâmica do metamodelo foi definida junto com a parte estrutural, não havendo uma distinção clara, à primeira vista, de quais são os elementos estruturais e quais são os dinâmicos. Isso porque a propriedade *isCurrent* (parte dinâmica) está definida juntamente com a propriedade *name* (parte estrutural) na

metaclasse *State*. Somente o leitor com conhecimento da semântica de execução desta linguagem conseguiria fazer essa distinção entre as partes.

Para que fique explícita esta distinção sem necessidade de interpretação sob a luz da semântica, [15] propõe um padrão de projeto (*design pattern*) para metamodelos de linguagens de modelagem executáveis. Nesse padrão, três pacotes (*packages*) MOF compõem o metamodelo e são unidos por associações de mesclagem (*merge*): *Domain Definition MetaModel* (DDMM), *State Definition MetaModel* (SDMM) e *Event Definition MetaModel* (EDMM). O primeiro pacote refere-se à parte estrutural, enquanto os dois últimos à parte dinâmica.

O SDMM engloba as propriedades necessárias para armazenar o estado de execução do modelo. Enquanto a propriedade *name* de *State* está no pacote DDMM, *isCurrent* fica em SDMM. A operação de mesclagem entre ambos gera uma metaclasse *State* completa com ambas as propriedades. Enquanto isso, o pacote EDMM especifica os eventos de execução que dirigem a execução dos modelos conformes a MM_L. Podem ser não apenas eventos de hardware, mas também eventos de software mais abstratos tais como eventos de leitura e escrita, eventos de comunicação para envio e recebimento de dados, notificações de relógios, etc. [15]. Os eventos concretos que ocorrem na execução de um modelo definem valores de propriedades dos elementos de EDMM. No exemplo da máquina de estado da Fig. 1, a metaclasse *Event* faria parte do pacote EDMM e é referenciada pela propriedade *event* de *Transition*.

Ademais, para modelos baseados em eventos discretos, o padrão estabelece que as metaclasses de evento de EDMM devem ser todas subclasses de uma metaclasse mãe abstrata, que abstrai o conceito de estímulo e que fica em um quarto pacote: *Trace Management MetaModel* (TM3).

O padrão também estabelece o pacote *Semantics* que carrega a semântica de execução nos modelos. Algumas considerações sobre esta parte serão vistas adiante. A Fig. 4 provê uma ilustração com visão geral sobre o padrão.

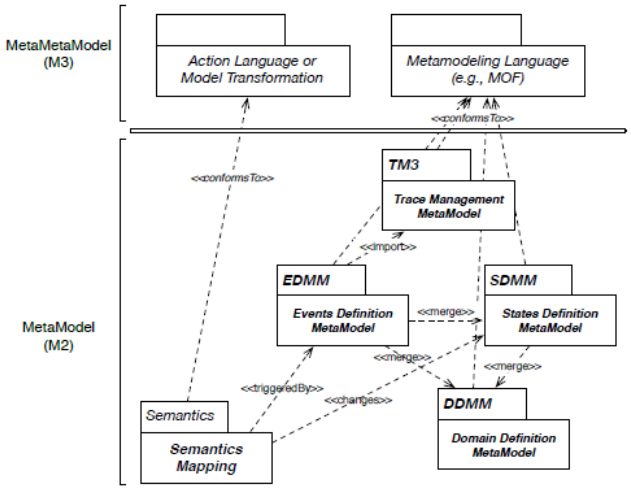

Fig. 4. Padrão de projeto para i-DSMLs. Fonte: [15].

B. Mapeamento de Modelos Executáveis em Dispositivos Não Adaptativos

O mapeamento de modelos executáveis em dispositivos não-adaptativos guiados por regras conforme (1) é apresentado através da descrição de um algoritmo com exemplos em cada passo. Como entradas, este algoritmo recebe o modelo executável que se deseja mapear e o metamodelo da i-DSML. Supõe-se que o metamodelo siga o padrão de projeto descrito na subseção anterior, ou seja, é possível distinguir os constituintes pertencentes a DDMM, SDMM e EDMM. Como saída, é produzido um dispositivo não-adaptativo $ND = (C, NR, S, c_0, A, NA)$.

ND inicia com *C*, *NR* e *A* vazios e c_0 nulo. *NA* e *S* iniciam como $\{\varepsilon\}$. Em seguida, uma configuração é gerada e incluída no conjunto *C* para cada combinação possível de valores das propriedades de estado (aquelas definidas em SDMM) para os elementos do modelo que a contém. As restrições definidas no metamodelo devem ser respeitadas.

No exemplo da máquina de estados, a propriedade *isCurrent* é a única definida em SDMM e pode assumir os valores *False* e *True*. Digamos que um modelo específico contenha três estados: *S1*, *S2* e *S3*, que são instâncias da metaclasse *State*. Essas instâncias assumem valores para *isCurrent*, e as combinações podem ser representadas por tuplas do tipo (*True, False, False*), (*False, True, False*), (*True, True, False*), etc. Uma restrição no metamodelo impõe que só pode haver o valor *True* para um único estado simultaneamente. Isso descarta a combinação (*True, True, False*), por exemplo, restringindo o conjunto de combinações na forma de tuplas que representam as configurações válidas do modelo em execução. Transições (instâncias de *Transition*) não são consideradas aqui pois não possuem propriedades definidas em SDMM, ou seja, não armazenam informações que definem o estado de execução do modelo.

No exemplo de redes de Petri, a propriedade *numberOfTokens* definida em SDMM representa o número de *tokens* contidos em cada posição em um dado momento de execução. Nesse caso, para uma rede com 3 posições, tuplas do tipo (*0,0,0*), (*1,2,0*), (*0,3,1*), etc. representam as configurações possíveis. É claro que se torna inviável criar configurações para cada combinação de inteiros possível, até porque o conjunto *C* seria infinito e isso não é permitido na definição de dispositivo não-adaptativo [3]. Nesse caso, deve-se impor restrições que limitem a faixa de operação do modelo conforme necessidade de aplicação, gerando um número de configurações finito condizente com a necessidade.

De maneira similar, geram-se as combinações de eventos concretos possíveis para popular o conjunto *S* de *ND*. Os eventos concretos do modelo executável são as instâncias das metaclasses definidas em EDMM preenchidas com cada combinação de valores de suas propriedades.

Seja um modelo de sistema de controle de velocidade de um trem, adaptado de [12] e apresentado na Fig 5. Na sintaxe concreta adotada para este modelo, as transições estão conectadas por linhas aos objetos de eventos que a disparam.

Assim, destaca-se que os eventos definidos no modelo são instâncias de metaclasses de EDMM.

O modelo apresentado na Fig. 5 é uma instância do metamodelo da Fig. 6, que consiste em uma especialização de máquinas de estado.

A três metaclasses de evento definidas em EDMM são *RedEvent*, *AmberEvent* e *GreenEvent* representando sinalizações luminosas. O primeiro tipo de evento indica que o trem deve parar. O segundo indica que deve trafegar a 40 km/h, enquanto o terceiro estabelece o nível normal de velocidade que pode ser 100 km/h ou 130 km/h. A sinalização *Green* é acompanhada por uma informação de qual velocidade o trem deve atingir. O sistema de controle do trem recebe essas três possíveis sinalizações do ambiente e ajusta sua velocidade de acordo. *RedEvent* e *AmberEvent* não possuem propriedades, enquanto *GreenEvent* possui a propriedade *speed* que pode assumir os valores 100 ou 130 nas instâncias. Cada sinalização é representada por uma instância de uma das três metaclasses, a saber: *Red*, *Amber*, *Green100* e *Green130*.

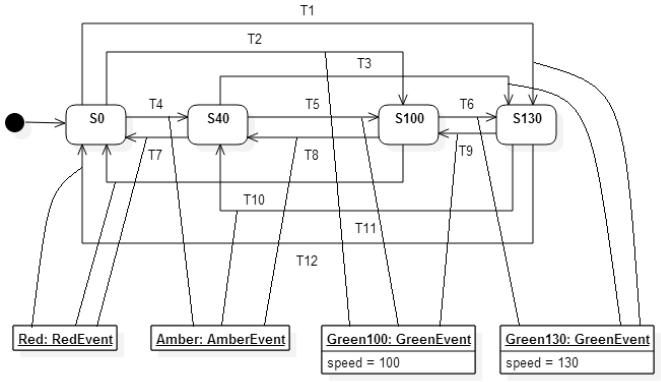

Fig. 5. Máquina de estados de sistema de controle de trem.

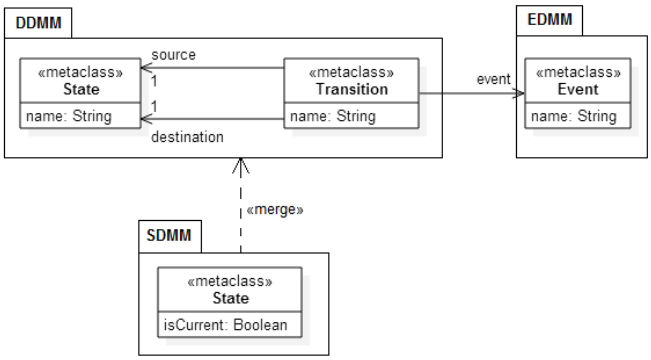

Fig. 6. Metamodelo (parcial) ao qual a máquina de estados de sistema de controle de trem é conforme.

O conjunto *NR* de *ND* deve ser montado de acordo com a semântica de execução do modelo. A semântica de um modelo executável é atrelada a um modelo de computação, ou *Model of Computation* (MoC). Considera-se para os fins deste trabalho um modelo de computação baseado em eventos discretos.

Uma das formas de se especificar essa semântica [15] é através de uma função de transição *f* que estabelece como o modelo evolui em sua execução. Esse modo reflete uma semântica operacional através da definição explícita das transições. Uma semântica translacional, por exemplo, traduziria os elementos da i-DSML em outra i-DSML com semântica conhecida. Deve-se supor que o algoritmo de mapeamento aqui proposto possua acesso a *f*, mesmo que tenha sido estabelecida de forma translacional, a tradução intermediária fica transparente para o algoritmo.

A função *f* toma como argumentos um modelo, uma representação do estado de origem, que pode ser a tupla que carrega os valores de propriedades de estado para cada elemento, e uma representação de evento concreto. É retornado o estado de destino do modelo na mesma representação do estado de origem.

No algoritmo de mapeamento, para cada combinação estado de origem × evento concreto do modelo, deve ser calculado o estado destino através de *f* e deve ser gerada uma regra a ser incluída em *NR* com os correspondentes objetos de configuração de origem, evento e configuração de destino do dispositivo não-adaptativo.

Deve-se frisar que *f* é definida com relação ao metamodelo, e não para modelos específicos. Por exemplo, para o metamodelo da Fig. 1, *f(M,descritor_estado, descritor_evento*) pode ser definida, resumidamente, através de um algoritmo que executa os seguintes passos: (i) localizar e armazenar na variável local *t* a instância da metaclasse *Transition* em *M* cuja propriedade *source* corresponda ao estado designado por *descritor_estado* e a propriedade *event* seja o estado designado por *descritor_evento*; (ii) obter o valor da propriedade *destination* de *t* e armazenar na variável *d*; (iii) retornar tupla de valores de propriedades de estado que corresponda a *isActive = False* para todos os estados exceto *d*, que deve corresponder a *True* (descritor do próximo estado). Dessa forma, *f* se aplica a qualquer máquina de estado conforme ao metamodelo.

Observa-se que a função *f*, que define a semântica de execução, "conhece" o metamodelo estruturalmente para determinar a próxima representação de estado com base no significado pretendido das abstrações. Por outro lado, sendo especificada desta maneira, em que os argumentos de estado e evento são descritores gerais aplicáveis a quaisquer modelos conformes a metamodelos que seguem o padrão de projeto, consegue-se padronizar também a assinatura *f(M, descritor_estado, descritor_evento*) → *descritor_estado* como forma de especificar uma semântica operacional.

O fato de *f* ser uma função implica que os modelos sejam determinísticos. Trabalha-se com esta hipótese por simplificação, mas se *f* for uma relação, então pode-se trabalhar também com modelos executáveis não determinísticos mediante algumas adaptações no algoritmo.

O padrão de projeto de [15] não especifica uma maneira padronizada de definir o estado inicial dos modelos executáveis. Essa característica seria importante para que o mapeamento proposto possa identificar o estado inicial de

qualquer modelo. Sendo assim, como os modelos são autocontidos por hipótese, vamos estabelecer que cada um deva, obrigatoriamente, definir um valor inicial (*default*) para cada propriedade proveniente de SDMM para cada uma de suas instâncias. No exemplo da Fig. 5, o estado inicial *S0* está indicado pela notação padrão de máquinas de estado. Porém, no modelo em si, essa situação é representada por *S0* possuir o valor inicial *True* para a propriedade *isCurrent*, enquanto os outros estados possuem *False*.

O algoritmo aqui apresentado é resumido em alto nível na Fig. 7, sem detalhar os laços de geração de combinações para fins de brevidade. Também não são detalhadas estruturas de dados auxiliares que mantêm correspondências entre elementos mapeados para uso ao longo do algoritmo.

O algoritmo da Fig. 7 não explora o conjunto *A* de estados de aceitação e nem o conjunto *NA* de símbolos de saída. Eles são sempre retornados como vazio e {ε}, respectivamente. Esta parte será deixada como possibilidade de extensão futura, a ser discutida na seção V.

C. Extensão do Padrão de Projeto para Adaptatividade

Embora o padrão de projeto proposto por [15] não preveja adaptatividade, é possível propor sua extensão através da inclusão de metaelementos que abstraem os conceitos necessários.

Propomos a inclusão de um pacote que represente a parte adaptativa do metamodelo da i-DSML descrita na seção II. Seguindo o padrão de nomenclatura, denominemos este pacote de *Adaptive Layer Definition MetaModel* (ALDMM).

Na extensão aqui proposta, o ALDMM é um pacote padronizado que pode ser aproveitado por todas as i-DSMLs. Ele define abstrações cujas instâncias especificam funções adaptativas, tomando como referência as ideias de [10] para modelagem de programas adaptativos, que é um tipo de dispositivo adaptativo. A Fig. 8 apresenta este pacote.

As decisões para a formulação dos metaelementos de ALDMM também foram baseadas nos conceitos apresentados pela seção III. Ou seja, usou-se o conhecimento mais geral da área de adaptatividade para estender o padrão de metamodelagem referido, incorporando explicitamente conceitos de adaptatividade. Os modelos executáveis das linguagens de modelagem que seguem este padrão ganham então a possibilidade de apresentar capacidades adaptativas, trazendo esta abordagem para o contexto da MDE.

A metaclasse *AdaptiveFunction* abstrai o conceito de função adaptativa contendo as propriedades *name* (nome simbólico), *parametersNames* (*array* de nomes de parâmetros), *generatorsNames* (*array* de nomes de geradores) e *body* (corpo da função). Cada instância desta metaclasse no modelo adaptativo representa uma função adaptativa que pode ser usada para compor ações adaptativas.

A metaclasse *AdaptiveAction* corresponde à abstração que representa as ações adaptativas, ou seja, chamadas de funções adaptativas. Cada instância desta metaclasse no modelo representa uma chamada de função adaptativa. Esta metaclasse está associada a *AdaptiveFunctionCallArgument* com multiplicidade 0..*.

```
Procedimento
Mapeia_Modelo_Executavel_em_Dispositivo_Nao_Adapt:

Entradas: M: Modelo executável
          MM: Metamodelo composto por DDMM,SDMM,
              EDMM, semântica operacional f
              e conjunto de restrições R.
Saídas:   ND: Dispositivo não-adaptativo guiado
              por regras = (C,NR,S,c0,A,NA)
Variáveis: config_candidata,c: tupla;
           lista_config_candidatas: conjunto;
           evento_candidato,s:  Inteiro;
           lista_eventos_candidatos: conjunto;
Início:

// Inicializações da saída:
C := conj. vazio; NR := conj. vazio;
S := {ε}; c0 := nulo;
A := conj. vazio; NA := {ε};

lista_config_candidatas :=
  Gerar lista de tuplas representando todas as
combinações de valores que representam estados de
execução do modelo. // Cada tupla codifica uma
configuração.

// Monta o conjunto C, adicionando apenas tuplas
// relativas a configurações que respeitam o
// conjunto R de restrições do metamodelo MM:
Para cada config_candidata em
          lista_config_candidatas {
   Se estado corresp. a config_candidata respeita R
   { C := C ∪ config_candidata; }
}

lista_eventos_candidatos :=
  Obter todos os elementos do modelo que são
instâncias de metaclasses provenientes de EDMM.
Cada uma será mapeada em um número inteiro (ex:
sequencial) que representará um símbolo permitido
na cadeia de entrada, e este inteiro será parte da
lista.

// Monta o conjunto S, adicionando apenas
// eventos que respeitam o conjunto R de restrições
// do metamodelo MM:
Para cada evento_candidato em
          lista_eventos_candidatos {
   S := S ∪ evento_candidato;
}

// Monta o conjunto NR
Para cada c em C {
   Para cada s em S {
      Se definido f(descritor de c,
                    descritor de s) {
         NR := NR ∪ (c,
                     s,
                     config. corresp. a
                     f(descritor de c,
                       descritor de s),
                     ε)
      }
   }
}

c0 := tupla representando a combinação de valores
iniciais das propriedades que representam estados
de execução do modelo;

Fim.
```

Fig. 7. Algoritmo em alto nível que mapeia modelos executáveis em dispositivos não-adaptativos guiados por regras.

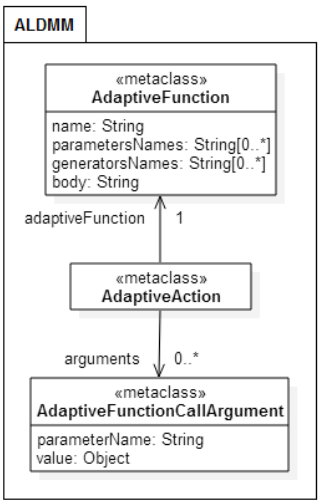

Fig. 8. Pacote ALDMM que estende o padrão de projeto de i-DSMLs executáveis para i-DSMLs adaptativas.

Cada instância dessa última metaclasse é um argumento que determina um valor para um dos parâmetros da função adaptativa chamada. A Fig. 9 mostra o padrão de projeto atualizado com a parte e semântica adaptativa.

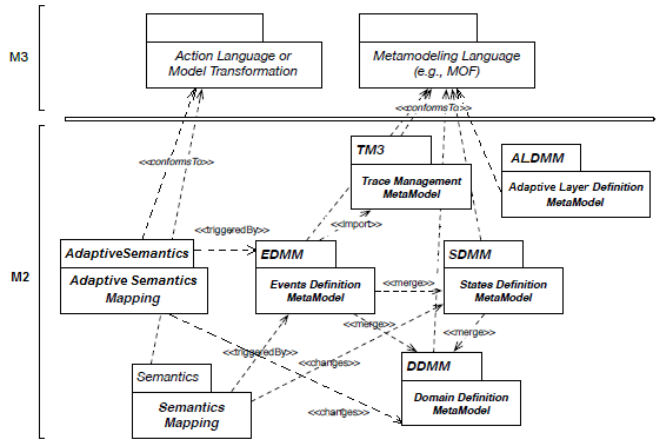

Fig. 9. Extensão do padrão de projeto para i-DSMLs adaptativa.

Conforme visto na seção II, também se faz necessária a introdução de uma semântica adaptativa para definir uma i-DSML adaptativa. Na subseção anterior apresentamos uma definição de semântica operacional através de uma função *f* que explicitamente define como o modelo executável evolui.

A semântica adaptativa pode ser estabelecida do mesmo modo por uma função *g* que toma os mesmos argumentos como entrada: um modelo, uma representação do estado de origem do sistema na forma de tupla que carrega os valores de propriedades de estado para cada elemento, e uma representação de evento concreto. Como saída, *g* fornece um par de ações adaptativas a serem aplicadas antes e depois da efetivação do passo de evolução do modelo executável. Como o pacote ALDMM tem a intenção de ser o mesmo para todos os metamodelos de i-DSMLs adaptativas, então dessa forma a assinatura da semântica operacional *g*(*M*, *descritor_estado*, *descritor_evento*) → (*ação_adaptativa_antes*, *ação_adaptativa_depois*) também fica padronizada, não sendo específica por metamodelo.

Uma constante *EmptyAdaptiveAction* representa uma ação adaptativa nula. Lembrando que ações adaptativas são instâncias de *AdaptiveAction* no modelo executável.

O corpo da função adaptativa é modelado como uma *string* através da propriedade *body* de *AdaptiveFunction*. Não há imposição aqui sobre a linguagem na qual ela deve ser definida. Para oferecer uma alternativa, citamos o formalismo SBMM [20], que define funções de edição de modelos genéricas, podendo ser usadas para modelos conformes a qualquer metamodelo.

- *insert_empty_instance(M, $name_i$, $name_c$)*: retorna um modelo atualizado construído pela inserção de uma nova instância nomeada por $name_i$ no modelo *M*, cuja metaclasse é aquela de nome $name_c$. Os valores iniciais de cada propriedade são os valores padrões definidos no metamodelo;
- *insert_or_edit_slot*(*M*, $name_i$, $name_p$, *val)*: retorna um modelo atualizado construído pela inserção de um novo *slot* [20] em *M*, i.e, atribuição de valor de uma propriedade a uma instância do modelo, ou atualização do mesmo caso o *slot* da mesma propriedade já exista. O *slot* será atribuído a instância de nome $name_i$, referente à propriedade de nome $name_p$ e terá valor *val*;
- *delete_slot(M, $name_i$, $name_p$)*: retorna um modelo atualizado com a remoção do *slot*, previamente adicionado, da instância $name_i$ de *M* referente a propriedade $name_p$, sendo de interesse para limpar valores atribuídos a propriedades em uma instância;
- *delete_instance(M, $name_i$)*: retorna um modelo atualizado com a remoção da instância de nome $name_i$ de *M*;
- *query_instance_by_prop(M, $name_p$, val)*: retorna um subconjunto de instâncias de *M* cujos *slots* da propriedade $name_p$ possuem *val* como valor atribuído.

Tais funções podem ser usadas como base para a construção do corpo de uma função adaptativa no contexto de modelos adaptativos.

Tendo em vista que as ações e funções adaptativas do modelo são definidas, pela extensão do padrão de projeto, como instâncias do mesmo modelo, mais especificamente das metaclasses *AdaptiveFunction*, *AdaptiveAction* e *AdaptiveFunctionCallArgument*, então as funções do SBMM aqui apresentadas também são capazes de alterar a parte adaptativa do modelo, provendo capacidade de meta-adaptatividade. No entanto, para fins do mapeamento pretendido, é imposta a restrição de que só podem ser utilizadas chamadas de funções adaptativas que alterem a parte estrutural do modelo, isto é, que criem ou removam instâncias de metaclasses de DDMM. Eventos concretos ou até mesmo ações adaptativas não podem ser criadas ou removidas pois o dispositivo adaptativo conforme formulado em [3] não permite a alteração do conjunto *S* nem *BA* ou *AA* em cada passo.

D. Mapeamento do Mecanismo Adaptativo

O algoritmo apresentado na subseção III.B é aqui invocado em um outro algoritmo que o completa, gerando também os

conjuntos *AR*, *BA* e *AA* do dispositivo adaptativo na forma de (2).

```
Procedimento
Mapeia_Modelo_Adaptativo_em_Dispositivo_Adapt:

Entradas: M: Modelo adaptativo
          MM: Metamodelo composto por DDMM,SDMM,
              EDMM, semântica operacional f,
              conjunto de restrições R, ALDMM e
              semântica adaptativa operacional g.
Saídas:   AD: Dispositivo adaptativo guiado
              por regras =
              (C,AR,S,c0,A,NA,BA,AA)
Variáveis: ND: Dispositvo não-adaptativo;
           par_aa,ar: tupla;
           lista_config_candidatas: conjunto;
           evento_candidato,s:  Inteiro;
           lista_eventos_candidatos: conjunto;
Início:

  // Inicializações da saída:
  AR := conj. vazio;
  BA := conj. vazio;
  AA := conj. vazio;

  ND :=
  Mapeia_Modelo_Executavel_em_Dispositivo_Nao_Adapt
   (M,MM)

  // Consideremos que ND = (C,NR,S,c0,A,NA) e
  // seus membros podem ser acessados por este
  // algoritmo.
  Para cada r = (c_src,s,c_dst,x) em NR {
     Se definido g(descritor de c_src,
                   descritor de s) {
        // par_aa = par de ações adaptativas
        // "antes" e "depois".
        par_aa := g(descritor de c_src,
                    descritor de s);
        // par_aa[1] é ação "antes".
        // par_aa[2] é ação "depois".
        BA := BA ∪ converte_aa(par_aa[1]);
        AA := AA ∪ converte_aa(par_aa[2]);
        AR := AR ∪ (converte_aa(par_aa[1]),
                    c_src,
                    s,
                    c_dst,
                    ε,
                    converte_aa(par_aa[2]));
     } Senão {
        AR := AR ∪ (ε,
                    c_src,
                    s,
                    c_dst,
                    ε,
                    ε);
     }
  }
Fim.
```

Fig. 10. Algoritmo em alto nível que mapeia modelos adaptativos em dispositivos adaptativos guiados por regras.

A função *converte_aa* usada no algoritmo abstrai a conversão de ações adaptativas como instâncias de *AdaptiveAction*, que seguem o padrão de projeto, para objetos que representam ações adaptativas dos conjuntos *BA* e *AA* de acordo com a formulação de dispositivos guiados por regras de [3].

V. Discussão

Este trabalho apresentou um mapeamento de modelos executáveis adaptativos para dispositivos adaptativos guiados por regras conforme a formulação de [3]. Para que isso fosse possível, foi necessário primeiramente reunir conceitos da literatura para estabelecer o entendimento sobre modelos executáveis e modelos adaptativos.

Como os graus de liberdade para se definir modelos executáveis são vários, buscou-se restringir o universo para modelos de processo autocontidos, conforme discussão da subseção II.A.

A estratégia geral adotada foi mapear modelos executáveis em dispositivos não-adaptativos e em seguida mapear a camada adaptativa dos modelos adaptativos para o mecanismo adaptativo dos dispositivos adaptativos guiados por regras. Para estabelecer o mapeamento, era necessário extrair de forma genérica, de qualquer modelo executável do universo considerado, as partes que descrevem seu estado e as partes que descrevem os eventos que causam a evolução do modelo no passo de execução. Para atingir esse objetivo, recorreu-se ao padrão de projeto de [15], que estabelece bem essa distinção através da separação em pacotes dos metaelementos estruturais, de estado e de eventos. Esse padrão de projeto refere-se ao nível dos metamodelos (e não ao nível dos modelos), e por isso também é chamado de padrão de metamodelagem [15].

Como o padrão mencionado não apresentava metaelementos para definir o estado inicial dos modelos conformes, foi assumido que as propriedades de estado definidas no metamodelo da i-DSML devam obrigatoriamente conter um valor inicial nos modelos, e esta combinação de valores define o estado inicial. Isso foi necessário para o mapeamento, já que a configuração inicial do dispositivo não-adaptativo é obrigatória. Os conceitos de símbolos de saída e de configurações de aceitação não foi explorado no mapeamento pois não existem no padrão apresentado. Uma sugestão de trabalho futuro é criar outra extensão do padrão de projeto que permita definir esses conceitos na i-DSML projetada e explorá-los no mapeamento. Mesmo sem eles, a executabilidade e adaptatividade dos modelos, foco deste trabalho, não é prejudicada.

Também para que o trabalho fosse possível, foi necessário estender o padrão de projeto para incluir a parte adaptativa e sua semântica, conforme previsto, mas não detalhado por [12]. O pacote ALDMM foi definido através de metaclasses que representam ações adaptativas e funções adaptativas. Recorreu-se ao formalismo SBMM [20], já que o MOF não estabelece operadores ou funções de edição de modelos, para se apresentar funções de edição de modelos que possam servir para especificar o corpo das funções adaptativas.

O mapeamento aqui apresentado não se propõe a ser o único possível, até mesmo por impor algumas restrições aos modelos de entrada e assumir que a i-DSML segue determinado padrão de projeto. Ele pode ser adaptado ou especializado conforme necessidade ou características da i-DSML em consideração.

Um dos problemas identificados é o alto número de combinações possíveis de preenchimento de propriedades de estado, principalmente quando números inteiros são utilizados para determinar o estado de execução do modelo. Este problema foi identificado no exemplo das redes de Petri e pode gerar muitas configurações no conjunto *C* do dispositivo não-adaptativo. A delimitação de valores permitidos conforme necessidade de operação prática pode limitar esse conjunto. No entanto, para fins conceituais, não há essa preocupação desde que *C* tenha um número finito de elementos.

VI. CONCLUSÃO

Modelos executáveis são de grande importância para a MDE na medida em que são uma das possibilidades de aplicação prática desta abordagem, consistindo em uma alternativa à transformação de modelos. A MDE é considerada uma das mais recentes mudanças de paradigma da Engenharia de Software [21]. Propõe trazer dois benefícios principais: o aumento do nível de abstração em relação à programação tradicional e a possibilidade de aproveitar os mesmos modelos de alto nível para gerar ou executar o sistema em várias plataformas.

Dentro do contexto de modelos executáveis, a adaptatividade é um recurso ainda pouco estudado [12] e, portanto, um campo a ser explorado. Por outro lado, o estudo da adaptatividade de forma geral já vem ocorrendo por diversos pesquisadores e apresenta resultados em várias áreas [4][8][9]. Sendo assim, estender o padrão de metamodelagem de x-DSMLs encontrado na literatura [15] para que as linguagens de modelagem incorporem capacidades adaptativas para seus modelos é de grande valia para ampliar os horizontes da modelagem executável. Além disso, colocar modelos executáveis no contexto mais amplo de dispositivos adaptativos guiados por regras permite aproveitar os resultados e o *toolbox* da teoria subjacente. Como exemplo, o algoritmo de operação de dispositivos adaptativos genéricos guiados por regras de [3] poderia ser aplicado para o desenvolvimento de motores de execução de modelos adaptativos com a utilização do mapeamento proposto.

Dessa forma, este trabalho contextualiza os modelos adaptativos dentro da área de pesquisa da adaptatividade, ampliando e conectando os conhecimentos. Com isso, busca-se reduzir o *gap* cultural e técnico entre as comunidades de metamodelagem/MDE e de teorias e modelos de computação.

Um possível trabalho futuro é a mudança da definição de semântica operacional de função para relação e a respectiva adaptação dos algoritmos para considerar modelos executáveis não determinísticos.

VII. REFERÊNCIAS

[1] OMG, MDA Guide Version 1.0.1, 2003. Disponível em: http://www.omg.org/cgi-bin/doc?omg/03-06-01.

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

[3] J. J. Neto, *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.

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

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

[6] Rutle et al., A formal approach to the specification and transformation of constraints in MDE. The Journal of Logic and Algebraic Programming 81, no. 4, 2012.

[7] S. R. M. Canovas and C. E. Cugnasca, C., "Um Metamodelo para Programas Adaptativos Utilizando SBMM". *WTA 2014, Oitavo Workshop de Tecnologias Adaptativas*, São Paulo, 2014.

[8] A. Contier, D. Padovani and J. J. Neto, "Linguístico: Usando Tecnologia Adaptativa para a Construção de Um Reconhecedor Gramatical". *WTA 2014, Oitavo Workshop de Tecnologias Adaptativas*, São Paulo, 2014.

[9] A. P. Silva, R. I. Silva Filho and F. A. Rodrigues, "Swarm Robotics: comportamento Adaptativo Aplicado ao problema de dispersão de pássaros em áreas agrícolas". *WTA 2014, Oitavo Workshop de Tecnologias Adaptativas*, São Paulo, 2014.

[10] S. R. M. Canovas and C. E. Cugnasca, "Em Direção ao Desenvolvimento de Programas Adaptativos Utilizando MDE", *WTA 2015, Nono Workshop de Tecnologias Adaptativas*, São Paulo, 2015.

[11] S. R. M. Canovas and C. E. Cugnasca, "Applying Model Driven Engineering to Develop a Bee Information System", in *Proc. EFITA*, Poznan, Poland, 2015, pp. 103-114.

[12] E. Cariou, O. Le Goaer, F. Barbier and S. Pierre, "Characterization of Adaptable Interpreted-DSML", in *European Conference on Modeling Foundations and Applications (ECMFA 2013)*, volume 7949 of LNCS, pp. 37-53.

[13] E. Breton and J. Bézivin, "Towards and understanding of model executability", in *Proceedings of the international conference on Formal Ontology in Information Systems (FOIS '01)*. ACM (2001).

[14] E. Cariou, C. Ballagny, A. Feugas and F. Barbier, "Contracts for Model Execution Verification", in *Seventh European Conference on Modeling Foundations and Applications (ECMFA 2011)*, volume 6698 of LNCS. Springer (2011).

[15] B. Combemale, X. Crégut and M. Pantel, "A Design Pattern to Build Executable DSMLs and associated V&V tools", in *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*. IEEE (2012).

[16] P. R. M. Cereda and J. J. Neto, "Utilizando linguagens de programação orientadas a objetos para codificar programas adaptativos", *WTA 2015, Nono Workshop de Tecnologias Adaptativas*, São Paulo, 2015.

[17] B. Combemale, C. Hardebolle, C. Jaquet, F. Boulanger and B. Baudry, *Bridging the Chasm between Executable Metamodeling and Models of Computation*. Software Language Engineering, 2013, pp. 184-203.

[18] E. Cariou, O. Le Goaer and F. Barbier, "Model Execution Adaptation?", in *7th International Workshop on Models@run.time (MRT 2012) at MoDELS 2012*.

[19] S. Pierre, E. Cariou, O. Le Goaer and F. Barbier, "A Family-based Framework for i-DSML Adaptation", in *European Conference on Modeling Foundations and Applications (ECMFA 2014)*, pp. 164-179.

[20] S. R. M. Canovas, *Uma proposta de formalismo para metamodelagem e suas aplicações na MDE*. PhD thesis, Escola Politécnica, Universidade de São Paulo, 2015. /no prelo/

[21] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Rath, D. Varró, M. Tisi and J. Cabot, "A research roadmap towards achieving scalability in Model Driven Engineering", in *Proceedings of the Workshop on Scalability in Model Driven Engineering*, pp. 2, ACM, 2013.

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 3. Vem desenvolvendo pesquisas sobre redes de controle, redes de sensores sem fio, eletrônica embarcada, aplicações de computação pervasiva e internet das coisas.