# 2
# Trabalhos Relacionados

Os trabalhos relacionados podem ser classificados em três categorias: abordagens baseadas em metamodelos para a definição de formalismos, uso de metamodelos em editores de diagrama e linguagens de regra, descritos nas subseções seguintes.

No presente trabalho, foram estudados também alguns editores de metamodelos propostos por diversos autores, a fim de servir de inspiração para a elaboração do editor aqui proposto. Em seu estudo, Minas [2006] propôs um editor de metamodelos que é baseado em edição livre pelo usuário (*free-hand*) ao mesmo tempo que suporta edição estruturada. Este editor é intitulado DiaMeta e é melhor descrito na seção de Ferramentas Existentes.



## 2.1
## Abordagens Baseadas em Metamodelos

O artigo de Gholizadeh & Azgomi [2010] apresenta uma abordagem sobre metamodelo para definição de um framework de modelagem multi-formal para redes de Petri e outras linguagens de modelagem formal relacionadas que podem ser representadas usando grafos, como as extensões de redes de Petri. A proposta do framework e da ferramenta relacionada é facilitar a inclusão de diferentes formalismos na ferramenta de uma maneira unificada.

Para a implementação dessa estrutura de meta-modelagem da ferramenta, o XML é utilizado como base da linguagem para toda a definição do metamodelo. Ele foi escolhido por alguns recursos, como XSD [XML Schema, 2010] e XSLT [XSLT, 2010], que estão disponíveis apenas nessa linguagem e combinam com o que era preciso para a definição do metamodelo. Também foi levada em consideração a existência de vários componentes já programados para documento XML, que foram utilizados no desenvolvimento da ferramenta para o framework. Em [Gholizadeh & Azgomi, 2009] há um maior detalhamento da definição formal do metamodelo.

Um modelo nesse framework é considerado como um modelo de classe e está organizado dentro das várias camadas da estrutura do metamodelo. As

classes do modelo não são solucionáveis e podem ser instanciadas para criar um modelo solucionável.

Como apresentado na Figura 1, existem quatro camadas dentro da estrutura do framework abordado. São eles:

1) Camada do meta-formalismo, que é a camada do topo e mais abstrata do framework.
2) Camada do formalismo, que é baseado no meta-formalismo.
3) Camada do modelo de classe, que pode ser definido baseado num formalismo que é definido dentro da camada do formalismo.
4) Camada do modelo, que inclui a solução final do modelo.

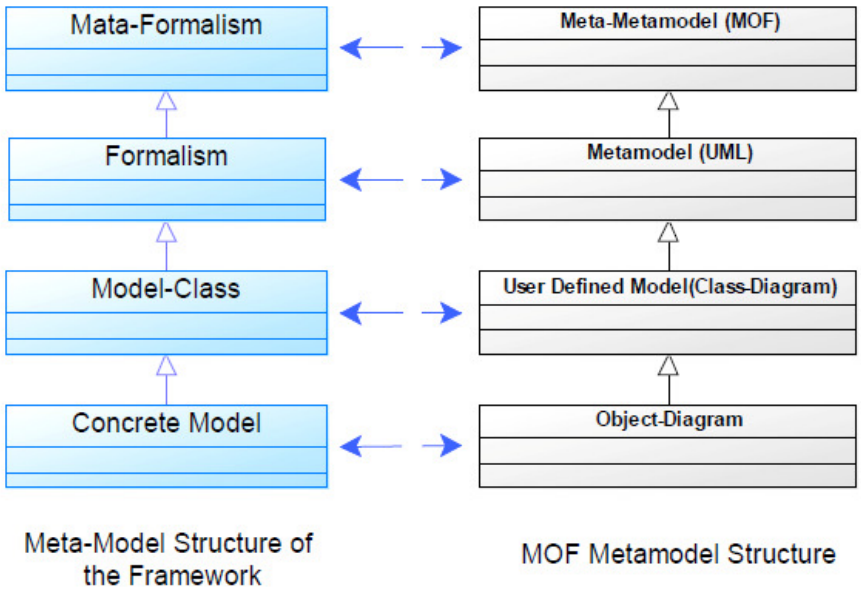


**Figura 1 - A estrutura do metamodelo do framework e seu mapeamento na estrutura do metamodelo MOF [Gholizadeh & Azgomi, 2010].**

As várias camadas da estrutura do metamodelo adicionam flexibilidade no formalismo do framework. Esse método torna o framework compatível com uma variedade de formalismos existentes, onde cada formalismo é definido baseado na sua definição de meta-formalismo. Por exemplo, para redes de Petri, a posição e transição dos elementos são definidos como nós e arcos são definidos como arestas do grafo. As notações gráficas, legendas e símbolos são definidas como propriedades de cada elemento da rede de Petri. A Figura 2 representa esse exemplo.

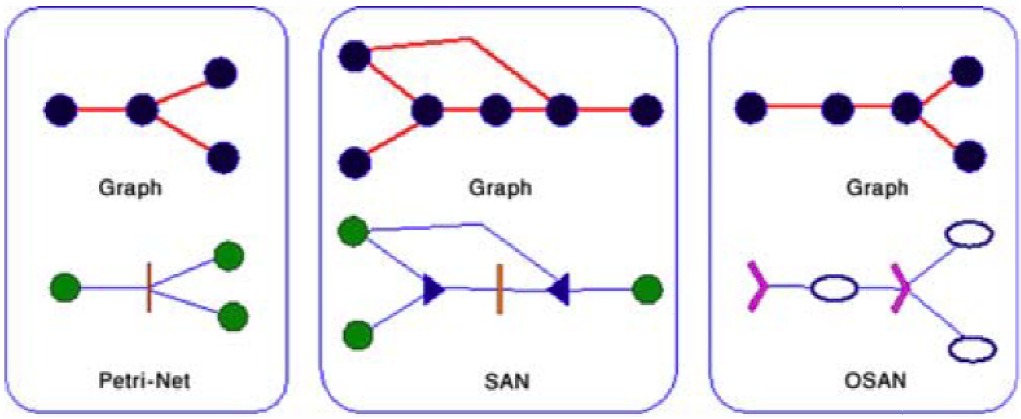


**Figura 2 - Exemplos de mapeamento dos elementos do formalismo para os elementos do grafo [Gholizadeh & Azgomi, 2010].**

Como na Figura 3 apresentada abaixo, a rede Petri é definida no framework obedecendo a estrutura do metamodelo. Cada elemento tem algumas propriedades a serem definidas como posição e imagem dentro do modelo, incluindo x, y, largura, altura e imagem, respectivamente.

```
<?xml version="1.0" encoding="UTF-8"?>
 <formalism image="petrinet.svg" name="Petrinet">
    <p name="eid" type="int"/>
    ……
    <p name="image" type="String"/>
    <element image="place.svg" name="Place"
      type="Node">
        <p name="eid" type="int"/>
        …
        <p name="token" type="int"/>
    </element>
    <element image="transition.svg"
      name="Transition" type="Node">
        …..
    </element>
    <element image="Arc.svg" name="Arc" type="Edge">
        <p name="eid" type="int"/>
        <p name="start" type="$Element"/>
        <p name="end" type="$Element"/>
        …….
        <ocl>
           context $Arc inv:(self.start=$Place
           implies self.end=$Transition) and
           (self.start=$Transition implies
             self.end=$Place)
        </ocl>
    </element>
</formalism>
```

**Figura 3 - Definição de uma rede Petri baseado na estrutura do metamodelo [Gholizadeh & Azgomi, 2010].**

No trabalho apresentado nesta dissertação, este artigo foi útil para mostrar quais caminhos devem ser seguidos para a construção da estrutura do metamodelo. Apesar do artigo ter um foco diferente do proposto nesta dissertação, o esquema XML apresentado nesse artigo foi de grande inspiração para a definição dos elementos e regras da linguagem.

## 2.2
## Editores de Diagramas Baseados em Regras

Maier & Minas [2009] propõe uma abordagem no desenho de diagramas visuais, que é aplicada tanto em metamodelos que permitem edição livre (*free hand*) quanto naqueles de edição estruturada. Essa aplicação é feita tanto dinâmica quanto estaticamente, durante a montagem dos metamodelos em um editor de metamodelos. Para a realização do trabalho, Maier utilizou a linguagem UML, porém, tal abordagem pode ser feita em outras linguagens como formulários GUI.

Maier propõe, em seu modelo, a separação entre duas partes: a primeira consiste na modelagem da informação abstrata da sintaxe do diagrama, e a outra consiste na modelagem da informação concreta da sintaxe do diagrama que é relevante para computar o layout. Esta segunda é o que representa as especificações que vão auxiliar a abordagem na reorganização do layout do metamodelo.

A ferramenta DiaMeta foi usada para exemplificar o método, que consiste em organizar os metamodelos durante sua edição, que pode ser feita tanto de forma estática, quando o usuário determina o momento da aplicação da abordagem (clicando em um botão), quanto dinâmica. Nesta última, a reorganização é feita durante o processo de criação do metamodelo, enquanto o usuário movimenta os elementos, levando em conta informações anteriores do modelo.

Ambos exemplos podem ser visualizados na Figura 4. A primeira demonstra a reorganização antes (a) e depois (b) da aplicação do layout estático.

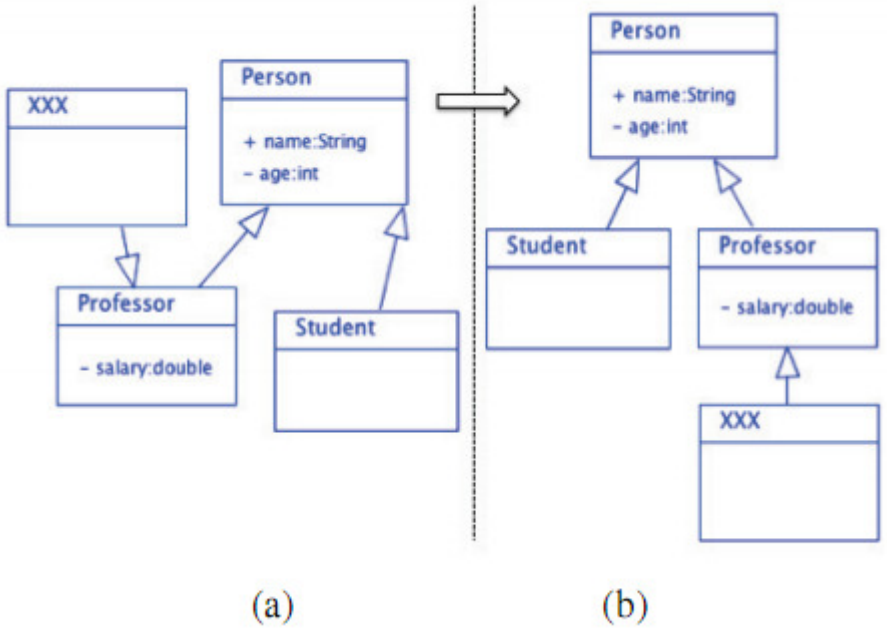


**Figura 4 - Antes (a) e depois (b) da aplicação do layout estático [Maier & Minas, 2009].**

Na figura 5 é possível visualizar a aplicação do layout dinâmico, antes (a) e após (b) o movimento da classe “Person”.

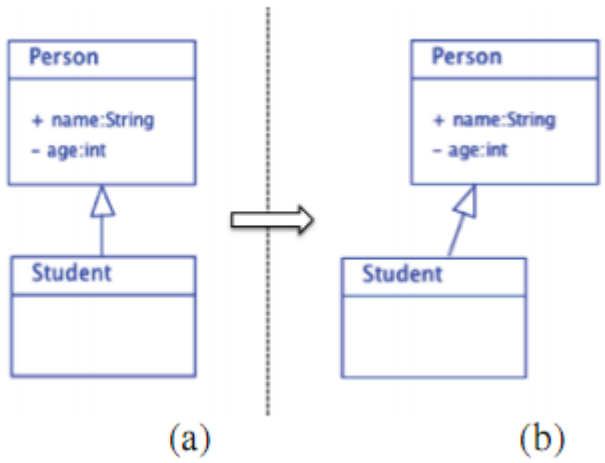


**Figura 5 - Antes (a) e depois (b) da aplicação do layout dinâmico [Maier & Minas, 2009].**

Para que tal abordagem seja aplicada, é preciso que haja, previamente, a especificação do layout desejado. As especificações do layout consistem em três partes: Layout Meta Model (LMM), um grupo de regras de layout e o controle de aplicação.

A aplicação de regras de layout foi um dos pontos mais importantes no artigo de Maier, a ser relacionado com o presente estudo, uma vez que a ferramenta proposta possui como uma de suas características a aplicação de

regras de sintaxe. Porém, o que difere a abordagem do artigo e o presente estudo é que na primeira, as regras são fornecidas pelo desenvolvedor da aplicação, enquanto que na ferramenta proposta, as regras são definidas pelo próprio usuário.

A regras são definidas baseadas no LMM. Elas são especificadas por mudanças nos atributos de componentes locais e são coordenadas pelo controle da aplicação.

Um exemplo de regra de layout mostrada no artigo, diz respeito ao tamanho do “Package” que contém as classes “Person” e “Student” (vistas na Figura 6 abaixo), em seu antes (a) e depois (b).

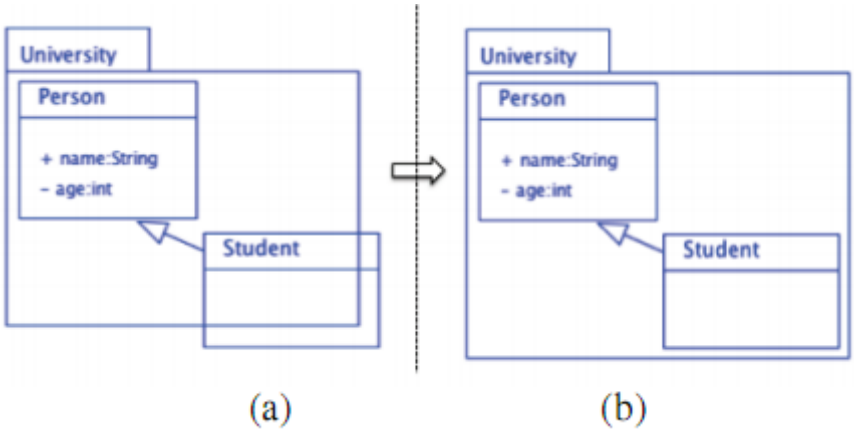


**Figura 6 - Antes (a) e depois (b) da aplicação da regra de layout [Maier, 2009].**

Para definir a regra, o desenvolvedor fornece padrão, condição opcional e uma ação. Neste caso, o padrão consiste em uma classe e um *package*. A classe cc1 foi o componente modificado pelo usuário. Neste exemplo, a condição será verificar se a classe movimentada permanece dentro do *package*. Caso contrário, a ação será atualizar os atributos x2 e y2 do *package* cp1, aumentando o tamanho do *package*. O ponto (x1, y1) corresponde ao canto superior esquerdo e (x2, y2) corresponde ao canto inferior direito. O conjunto de regras pode ser visualizado na Figura 7, separado em padrão (a), condição & ação (b).

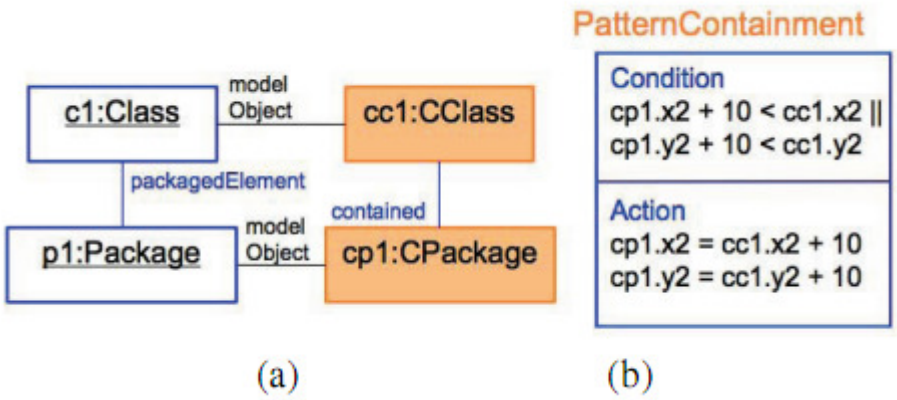


**Figura 7 - Conjunto de regras por padrão (a) e condição & ação (b) [Maier, 2009].**

## 2.3
## Linguagem de Regras

RuleML (Rule Markup Language) é uma linguagem de marcação com a iniciativa de criar e padronizar uma linguagem aberta e independente baseada em XML e RDF para regras [RuleML, 2011]. Ela foi desenvolvida de uma maneira modular e hierárquica para diferentes tipos de regras. A RuleML não tem intenção de ser executada diretamente, mas sim transformada em outra linguagem alvo baseada em regras.

Uma breve explicação de como seria uma linguagem baseada em RuleML [RuleML Datalog, 2011]. A RuleML Datalog é uma linguagem de marcação que representa informações relacionais onde cada coluna é uma frase em linguagem natural. Considerando a seguinte frase: "Peter Miller's spending has been min 5000 euro in the previous year". Ela pode ser transformada para a seguinte Datalog RuleML:

```
<Atom>
    <Rel>spending</Rel>
    <Ind>Peter Miller</Ind>
    <Ind>min 5000 euro</Ind>
    <Ind>previous year</Ind>
</Atom>
```

O verbo “spending” é rotulado como relacionamento e as três frases "Peter Miller", "min 5000 euro", e "previous year" são rotuladas como constantes individuais que são três argumentos do relacionamento, respeitando

a mesma sequência. O relacionamento completo da aplicação constitui uma fórmula atômica, por isso o rótulo “Atom”. A Figura 8 a seguir apresenta essa organização em um tipo de árvore de análise.

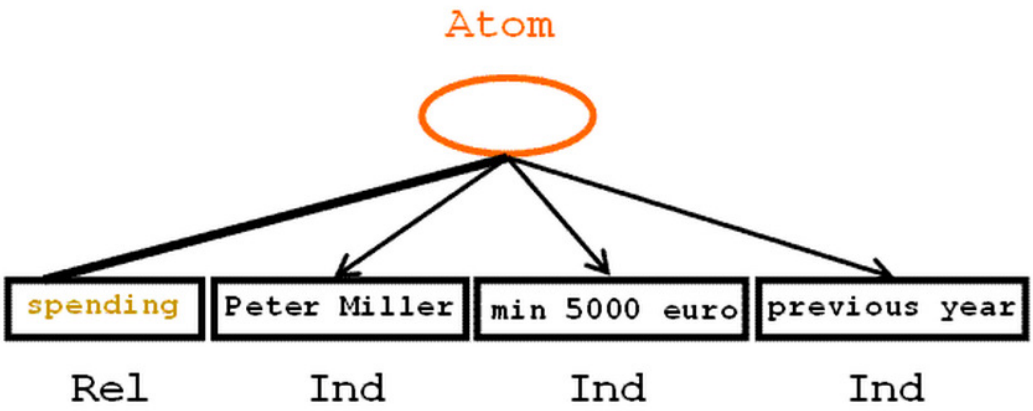


**Figura 8 - Árvore de análise entre os rótulos [RuleML Tutorial, 2011].**

Existem vários artigos relacionados à RuleML, que abordam a criação de regras de sintaxe a partir da linguagem RuleML, porém nenhuma é aplicada em linguagens visuais. Existem combinações de RuleML com outras linguagens como OCL e SWRL, apresentadas em [Wagner *et al,*2006], que integra linguagens de marcação para permitir o mapeamento de mais linguagens de construção sem a necessidade de tradução entre os diferentes tipos de expressões. Isso permite o aumento da usabilidade do intercâmbio entre os diferentes formatos de linguagem.

Paschke (2005) criou a linguagem RBSLA (Rule Based Service Level Agreement) baseada na RuleML. Com essa linguagem, é possível implementar o contrato de nível de serviço de um sistema que lê este tipo de sintaxe executando regras contratuais automaticamente.

Nossa linguagem aproveitou as seguintes ideias inspiradas na RuleML:a utilização de linguagem de marcação, a hierarquia rótulos e a utilização da negação.

## 2.4
## Ferramentas Existentes

Nesta seção são apresentados alguns editores de metamodelos existentes. Foi realizado um levantamento dessas ferramentas e suas principais funcionalidades, a fim de se obter um aprendizado acerca das funcionalidades mais interessantes para servir de inspiração para a construção do software proposto no presente trabalho. Além disso, também foi feito um levantamento das limitações desses softwares, que serviram de motivação à criação de um novo software.

### 2.4.1
### Microsoft Office Visio

O programa Visio da Microsoft [Microsoft, 2010] é a ferramenta que mais se aproxima do objetivo deste trabalho. Ele trabalha com vetores gráficos na criação e visualização de informações diversas. A Figura 9 seguinte apresenta a sua interface.

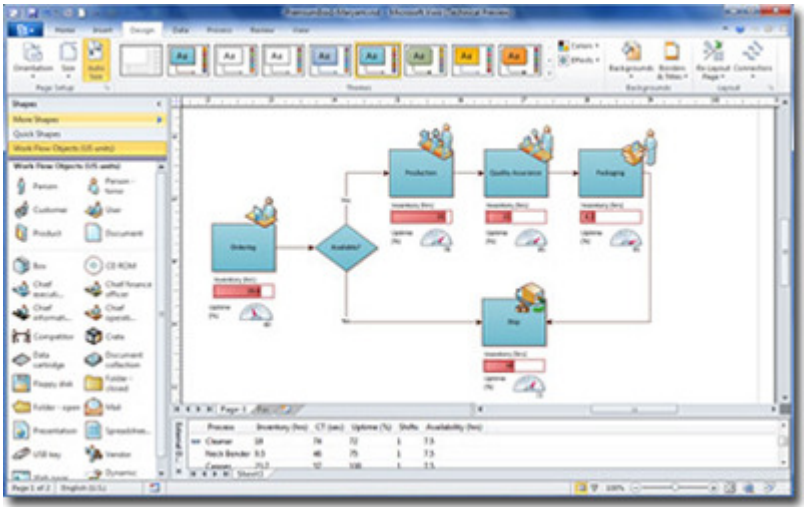

**Figura 9 - Programa Microsoft Visio 2010.**

Uma funcionalidade interessante do Visio é a importação e exportação de stencils. Um stencil é uma coleção de formas associada a um determinado modelo do Visio. O usuário pode criar um novo stencil para manter as formas que usa frequentemente e que pretende encontrar rapidamente.

A validação de diagramas, que é um dos objetivos do trabalho, também está presente no Microsoft Visio 2010. Entretanto, a criação de regras é bem complicada para uma pessoa pouco experiente em programação. Devido ao fato das regras serem elaboradas através da linguagem Visual Basic, cada regra exige a criação de um método de programação. A Figura 10 seguinte apresenta como seria a inclusão de uma regra no metamodelo.

```
Public Sub AddRuleSet()
' Add a validation rule set to the document.
' Edit the nameU to suit.
Dim doc As Visio.Document
Dim ruleSet As Visio.ValidationRuleSet
Dim nameU As String
    nameU = "Fault Tree Analysis"
    Set doc = Visio.ActiveDocument
    ' Check whether the rule set already exists.
    Set ruleSet = getRuleSet(doc, nameU)
    If ruleSet Is Nothing Then
        ' Create the new rule set.
        Set ruleSet = doc.Validation.RuleSets.Add(nameU)
    End If
    ruleSet.description = "Example Fault Tree Analysis rule set."
    ruleSet.Enabled = True
    ruleSet.RuleSetFlags = Visio.VisRuleSetFlags.visRuleSetDefault
End Sub

Private Function getRuleSet(ByVal doc As Visio.Document, _
    ByVal nameU As String) As Visio.ValidationRuleSet
' Return a named rule set or nothing.
Dim retVal As Visio.ValidationRuleSet
Dim ruleSet As Visio.ValidationRuleSet
    Set retVal = Nothing
    For Each ruleSet In doc.Validation.RuleSets
        If UCase(ruleSet.nameU) = UCase(nameU) Then
            Set retVal = ruleSet
            Exit For
        End If
    Next
    Set getRuleSet = retVal
End Function
```

**Figura 10 - Adição de uma regra no Visio**

O objetivo desse trabalho é oferecer uma solução mais simples ao usuário, algo que seja de fácil entendimento e sem necessidade de possuir um alto grau de conhecimento da linguagem de criação de regras. Sendo assim, optou-se por uma linguagem declarativa para a definição das regras dos metamodelos.

### 2.4.2
### Visual Library

Essa API de desenho foi concebida inicialmente como parte da IDE de desenvolvimento NetBeans [NetBeans, 2008]. Ela foi utilizada para exibição de diagramas na ferramenta *Dependencies Viewer* na tese de Couto [Couto, 2009]. Sua preferência por essa ferramenta foi motivada por ser orientada a metamodelos e sua predisposição para trabalhar com um produto voltado para a engenharia reversa. Essa ferramenta, criada na linguagem JAVA, foi desenvolvida para manipular diagramas e elementos gráficos. A biblioteca é de código aberto e pode ser usada gratuitamente, sem restrições. Esta API foi utilizada durante um tempo como uma solução candidata para a criação do editor de metamodelos, porém, no decorrer do desenvolvimento ela foi trocada por outra API chamada JgraphX que atendia melhor as expectativas do trabalho proposto.

A Visual Library (Figura 11) atende a projetos que têm como objetivo apenas apresentar, através de grafos, resultados de dados já computados por outra parte do sistema. Ao contrário da biblioteca JgraphX, a Visual Library não permite uma manipulação de elementos gráficos no nível requerido para a construção de um editor de diagramas. Por exemplo, o arraste de elementos da biblioteca para o diagrama ou a criação de arestas a partir do vértice com o arrastar do mouse. Por isso, para evitar o retrabalho de criação de algumas funcionalidades, a ferramenta JgraphX foi adotada no lugar da Visual Library. A troca foi um resultado positivo pois acelerou o processo de criação da ferramenta.

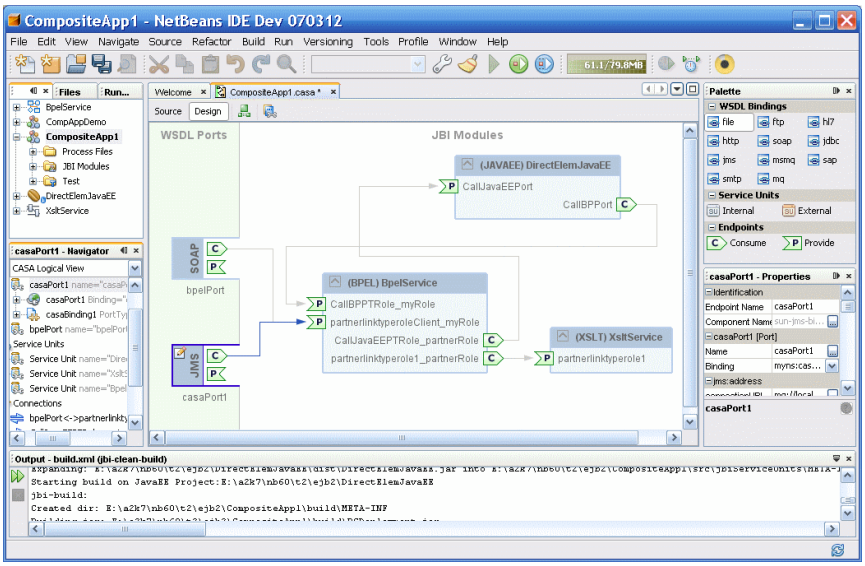


**Figura 11 - Exemplo de programa que usa a API Visual Library.**



### 2.4.3 JgraphX

JgraphX é uma biblioteca feita a partir de várias tecnologias que provê uma seleção de funcionalidades para tratar de diagramas interativos e grafos. Sendo uma biblioteca para desenvolvedores, ela não é uma aplicação pronta para usar, mas sim uma ferramenta para auxiliar em tarefas como: desenhar, interagir e associar um contexto (cenário ou tarefa) com um diagrama apresentado. A ferramenta é gratuita na versão Java para desenvolvimento desktop e paga na versão Javascript para desenvolvimento web. Na versão paga, o nome foi alterado para mxGraph, e é compatível com as tecnologias PHP, .NET e Java para web [JgraphX, 2011].

Essa foi a biblioteca escolhida para se integrar ao editor dirigido por metamodelos. Simplifica ações como a manipulação de objetos na tela e a associação de contexto do diagrama. Outro fator importante para a decisão da adoção dessa biblioteca é o suporte a gráficos vetoriais escalonáveis. Com a ajuda do programa Dia [Dia, 2011], será possível criar gráficos vetoriais escaláveis [SVG, 2010] para serem utilizados nos elementos do metamodelo. Com a função de exportação do programa Dia, é possível criar arquivos no padrão aceito pela biblioteca para funcionar com o editor. Esse padrão é a

combinação dos arquivos SHAPE e PNG: SHAPE para o gráfico vetorial escaláveis e o PNG para o ícone representante do elemento na biblioteca. Também será possível além dos gráficos vetoriais escaláveis, o uso de imagens JPEG, PNG e GIF.

### 2.4.4 Dia

Inspirado no programa comercial Microsoft Visio, o Dia [Dia, 2011] (Figura 12) é mais voltado para diagramas informais de uso casual. Ele pode ser usado para desenhar diferentes tipos de diagramas: diagramas de UML, fluxogramas, diagramas de rede, diagrama entidade relacionamento e muitos outros. Possibilita a exportação de inúmeros formatos de arquivo como: EPS, SVG, XFIG, WMF, PNG e SHAPE.

Neste trabalho, o aplicativo Dia é utilizado para auxiliar na criação dos elementos do metamodelo. Sua funcionalidade de exportação de arquivos SHAPE para a utilização no editor de diagramas tornou o programa essencial para se trabalhar com gráficos vetoriais escaláveis. Já que na documentação da biblioteca JgraphX é aconselhado o uso deste programa para a criação de novos elementos gráficos.

Utilizando a exportação do programa Dia, é gerado um arquivo SHAPE e um arquivo PNG. O arquivo SHAPE é na verdade um arquivo SVG (*Scalable Vector Graphic*) modificado com algumas informações extras. Já o arquivo PNG é uma imagem miniatura do SHAPE, com o mesmo nome, para ser utilizado na biblioteca de elementos do editor de diagramas.

Essa exportação é utilizada pelo editor de diagramas proposto nesse estudo para a geração dos elementos do metamodelo. O editor está preparado para receber os arquivos SHAPE e PNG exportados do Dia. Basta incluir a referência do elemento no arquivo XML do metamodelo.

É possível além do uso de arquivos SHAPE no editor, a utilização de imagens do tipo JPEG, PNG, GIF. Caso haja a necessidade de utilizá-los o programa Dia também exporta os elementos nestes formatos. Na seção 4 há uma explicação mais detalhada a respeito.

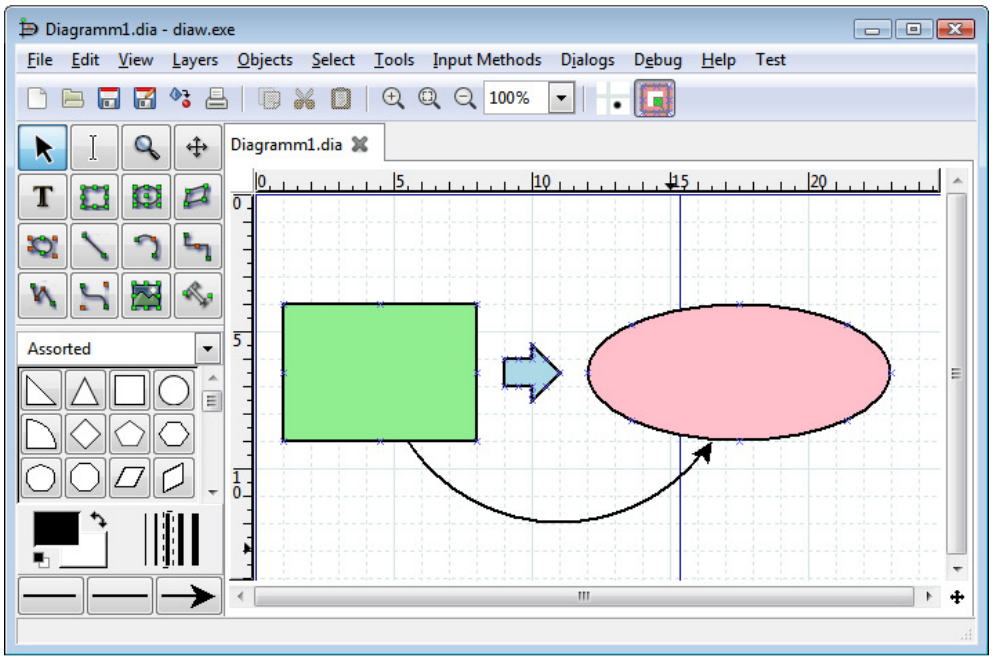


**Figura 12 - Interface do programa Dia.**



### 2.4.5 DiaMeta

Proposto por Minas [2006], esta ferramenta tem como principal objetivo oferecer uma maior liberdade ao usuário durante a criação e edição de metamodelos, uma vez que permite a edição livre (*free hand*) sobre os modelos. Além disso, ela também dispõe da opção de edição estruturada, possibilitando a integração de ambas as formas de edição. Com a edição livre, a ferramenta é capaz de identificar a corretude dos dados editados livremente pelo usuário, traduzindo estes dados, caso corretos, para um objeto estruturado.

A relação do editor DiaMeta (Figura 13) com o proposto nesse estudo é que os dois são editores de metamodelo e verificam regras. Entretanto a diferença do DiaMeta é a sua abordagem sobre regras estruturais da linguagem, onde o foco é a arrumação visual do diagrama. Já o editor proposto aborda as regras de sintaxe da linguagem, onde o usuário é alertado sobre os erros gramaticais do diagrama.

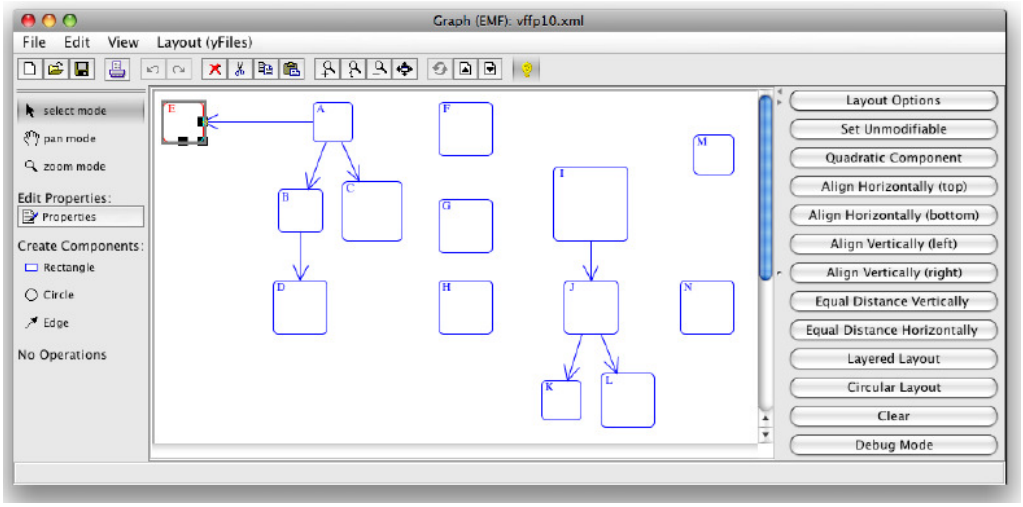


**Figura 13 - Interface do editor de diagramas DiaMeta [Minas, 2006].**