PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 24/08

Transformando o Diagrama de Atividade em uma Rede de *Petri*

Luana Lachtermacher

Denis Silva da Silveira

Rodrigo de Barros Paes

Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 24/08 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Abril, 2008

Transformando o Diagrama de Atividades em uma Rede de *Petri* através do QVT.

Luana Lachtermacher, Denis Silva da Silveira, Rodrigo de Barros Paes e Carlos José Pereira de Lucena
lulachter@gmail.com, denis@inf.puc-rio.br rbp@les.inf.puc-rio.br, lucena@inf.puc-rio.br.

Abstract With the evolution of the Model Driven Architecture (MDA), there is a big concern about model transformation. The OMG (*Object Management Group*) recently adopted a standard transformation language called Query View Transformation (QVT). The main goal of this paper is to perform a transformation between models, where the source model used was Activity Diagram shown in UML 2.0 and the target model was *Petri* Net.

Keywords QVT, Transformation model to model, Activity Diagram and *Petri* Net.

Resumo. Com a evolução do conceito de Arquitetura Dirigida a Modelo (MDA), existe uma grande preocupação sobre a transformação entre modelos. A OMG (*Object Management Group*) recentemente adotou um padrão para linguagem de transformação chamado Query View Transformation (QVT). Este artigo tem como objetivo fazer a transformação entre modelos, o modelo fonte utilizado foi o Diagrama de Atividades, apresentado na UML 2.0, e o modelo alvo foi a Rede de *Petri*.

Palavras-chave: QVT, Transformação modelo-modelo, Diagrama de Atividades, Rede de *Petri*.

Responsável por publicações
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Sumário

1 Introdução

A expressão *processo de negócio* pode ser definida como um conjunto estruturado de atividades ordenadas, no tempo e no espaço, que produzem um produto ou um serviço, com início e fim, e entradas e saídas bem definidas (HAMMER e CHAMPY, 1994). Sua implantação, no contexto de uma organização, consome normalmente uma grande quantidade de recursos (meios de produção: humanos, financeiros, matérias-primas, e equipamentos). Logo, é de grande utilidade para as organizações o uso de alguma metodologia que possa identificar, o quanto antes, as inconsistências nos processos de negócio. A execução desses processos inconsistentes irá acarretar um custo muito maior para as organizações que o do desenvolvimento da metodologia.

A modelagem de um processo de negócio pode ser feita de muitas formas. O'NEILL e SOHAL (1999) apresentam um conjunto variado de técnicas (métodos e ferramentas) para essa funcionalidade. No contexto do desenvolvimento de Sistemas de Informação orientados a objeto ou baseados em componentes, o Diagrama de Atividades tem ganho bastante destaque.

Por outro lado, as redes de *petri* pelo seu formalismo com base matemática e sua representação gráfica vem sendo amplamente utilizada para simulação, há mais de 30 anos, em vários domínios de aplicação, entre os quais se destacam os sistemas de manufatura, de transporte, logísticos e, de forma geral, todos os sistemas e eventos discretos (MURATA, 1989). No entanto, ainda apresenta pouca familiaridade para os projetistas de Sistemas de Informação quando comparada aos modelos da UML.

Sendo assim, a presente pesquisa desenvolverá um processo de transformação entre o Diagrama de Atividade, usado para modelagem de processos de negócio, para uma Rede de *Petri.* A motivação para tal está no fato de ser mais fácil para os projetistas de negócio, modelar o seu processo através do Diagrama de Atividade do que através das Redes de P*etri*. Entretanto, esse processo de negócio, quando transformado em uma Rede de *Petri*, em função do seu formalismo matemático, possibilitará a verificação de propriedades, conforme apresentado em MURATA (1989). Entre essas propriedades destacamos as de alcançabilidade, limitação, segurança, cobertura, limitação estrutural, repetitividade e consistência, que não são disponível em um Diagrama de Atividade. Para a realização desta transformação, foi utilizado a linguagem de transformação definida pelo OMG (*Object Management Group):* a Consulta/Visão/Transformação (Query/Views/Transformation, QVT) (OMG, 2004).

1.1 Objetivo

O objetivo deste trabalho foi fornecer uma visão geral sobre o QVT, apresentando, um exemplo na definição e execução da transformação do Diagrama de Atividades em uma Rede de *Petri*. Para alcançar esse objetivo, objetivos intermediários foram alcançados, como o estudo do QVT; do Diagrama de Atividades; da Rede de *Petri* e as ferramentas utilizadas para realizar a transformação, como por exemplo: *Eclipse Modeling Framework*, para fazer a geração dos metamodelos e do exemplo que será transformado e o *MediniQVT* para fazer a transformação em si, utilizando como entrada os dois metamodelos, o exemplo de Diagrama de Atividade e um *script* QVT desenvolvido.

Este trabalho irá primeiramente mostrar uma visão geral do Diagrama de Atividades, Rede de *Petri e* QVT. Depois é apresentado todos os passos para a transformação e uma conclusão do trabalho.

2 Diagrama de Atividade

A UML (BOOCH *et al.*, 2005) é um dos mais importantes padrões mantidos pelo grupo OMG. A UML vem sendo considerada como a linguagem de modelagem (gráfica) padrão no desenvolvimento orientado a objetos, oferecendo apoio à criação de modelos independentes da plataforma, nos quais os conceitos são separados da semântica existente nos modelos da implementação. As linguagens gráficas de modelagem existem há muito tempo na indústria de *software*, e o grande propulsor por trás de todas elas é o fato das linguagens de programação não possuírem um nível de abstração suficientemente alto que permita aos programadores raciocinar diretamente com os conceitos envolvidos num projeto (FOWLER, 2004).

Modelagem, no sentido mais amplo, é o uso econômico de algo (o modelo) no lugar de alguma coisa real, tendo em vista algum objetivo cognitivo. Esta prática permite o uso de algo mais simples, seguro e barato do que o sistema real, para o estudo do objetivo desejado(BOOCH,1994)(MEYER,1988). Um modelo é, portanto, uma representação simplificada de algum conceito ou situação, com os objetivos de sua observação, manipulação e entendimento (MELLOR *et al.*, 2004). No desenvolvimento de *software*, tal como em outras aplicações, os modelos são criados com o objetivo de diminuir a complexidade inerente aos temas de suas aplicações.

O Diagrama de Atividades é utilizado para descrever lógica de programação, processos de negócio e *workflows*. Este diagrama determina as regras essenciais de seqüência que se deve seguir para a execução do processo. Neste trabalho, usamos uma versão simplificada dos elementos do Diagrama de Atividades da UML 2.0 (OMG, 2005).

3 Rede de *Petri*

Rede de *Petri* é uma ferramenta de modelagem aplicável a uma série de sistemas, especialmente aqueles com eventos concorrentes. Elas foram criadas por *C. A. Petri*, um pesquisador alemão, que na sua tese de doutorado apresentou um tipo de grafo orientado e bipartido com estados associados, com o objetivo de estudar a comunicação entre autômatos (*PETRI*, 1961).

Um grafo $G(P, E)$ consiste de um conjunto vértices $P = \{p_1,..., p_n\}$, com $n > 0$, e de um conjunto de arcos $E = \{e_1,..., e_m\}$, tal que cada vértice é conectado com pelo menos outro vértice, e cada arco conecta dois vértices de P. O grafo G é dito orientado se todo arco possui um sentido, bem como é denominado finito se m e n são finitos. Um grafo, por sua vez, é considerado bipartido, quando o seu conjunto de vértices P pode ser particionado em dois conjuntos X e Y. Tal que toda aresta E de G é incidente a um elemento de X e a um elemento de Y (BOAVENTURA NETTO, 2006).

Como ferramenta matemática e gráfica, as redes de *petri* oferecem um ambiente uniforme para a modelagem, análise formal e simulação de sistemas e eventos discretos, permitindo uma visualização simultânea da sua estrutura e comportamento. No entan-

to, mais especificamente, as redes de *petri* modelam dois aspectos: eventos e condições. Assim como, as relações entre eles (HEUSER, 1990).

Os elementos dos dois conjuntos em que se podem dividir os vértices que constituem uma Rede de *Petri* denominam-se: lugares e transições. Os *lugares* são normalmente representados por circunferências ou elipses, e as *transições* por segmentos de reta, retângulos ou barras. Os lugares encontram-se ligados às transições, e estas aos lugares, através de arcos orientados, conforme ilustrado na.

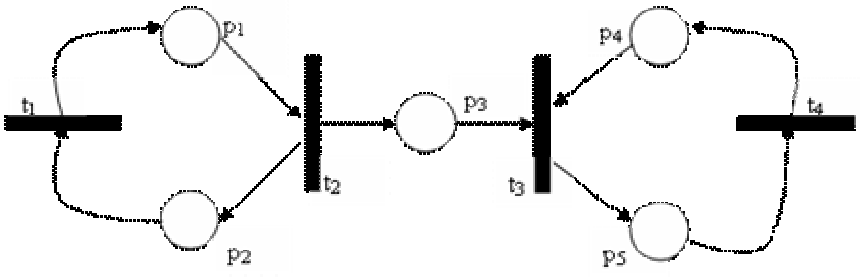

Figura 1 – Rede de *Petri*

Uma forma muito genérica e muito próxima as sua visualização gráfica, ser vista como depósitos de recursos e as transições como ações que manipulam esses recursos. Os recursos são representados graficamente por pequenos círculos pretos dentro dos lugares. Cada um desses círculos dá-se o nome de marca. Os lugares surgem como representantes do estado da rede e as transições como responsáveis pela mudança de estado. Desta forma, a Rede de *Petri* é simultaneamente orientada para os estados e para as ações. Ao *estado* da rede é usual chamar-se marcação, dado ser definido pela marcação de cada um dos seus lugares, ou seja, pela quantidade de marcas presentes em cada lugar.

A função das transições consiste em destruir e/ou criar marcas. Como as transições estão obrigatoriamente entre lugares é através da sua ação, denominada de disparo, que um lugar altera a sua marcação. Os arcos indicam, para cada transição, os lugares sobre os quais estas atuam. Este tipo de diagrama tem ajudado muito aos desenvolvedores para fazer a simulação de uma rotina e também vem sendo muito utilizado para modelar comportamento em sistema distribuídos.

4 Query View Transformation (QVT)

O QVT representa uma tentativa de se encontrar uma linguagem padrão para expressar transformações entre modelos, de uma forma simples e tendo por fim a sua execução automática (TRATT, 2003). A sigla QVT representa três conceitos:

- *Consulta* (*Query*), pegam um modelo como entrada e selecionam elementos específicos deste modelo;
- *Visão* (*View*), são modelos que são derivados de outros modelos, uma visão é uma projeção de um metamodelo que pode ser gerado a partir de consultas em um modelo UML e respectivas transformações;
- *Transformação* (*Transformation*), tomam um modelo como entrada e atualizam ou criam um novo modelo.

Apesar de o QVT estar dependente destes três conceitos o da transformação é o mais importante para esse trabalho. A seguir será mostrado todos os aspectos importantes da transformação utilizando QVT.

4.1 Arquitetura do QVT

O QVT tem uma arquitetura hibrida, sendo uma parte declarativa e a outra imperativa. As duas partes têm uma linguagem especifica.

A parte declarativa é utilizada para fazer todo o processo de transformação. A linguagem associada a esta parte, descreve os relacionamentos entre as variáveis utilizando funções ou regras de inferência. Para a execução desta linguagem é necessário um compilador ou interpretador, que aplica um algoritmo sobre as relações e produz um resultado. Esta camada pode ter informações suficientes para descrever uma transformação bidirecional ou unidirecional. (GARDNER *et al.,* 2002) Existem duas camadas declarativas. São elas:

A primeira camada é a *Relação* (*Relation)* que tem como objetivo fazer uma especificação do relacionamento entre modelos MOF. A linguagem utilizada nesta camada possibilita a equivalência entre de objetos complexos e cria classes de rastreamento para verificar o que ocorreu durante a transformação. Além da execução propriamente dita, uma relação pode ser utilizada apenas para garantir que a outra relação está correta.

A segunda camada é chamada de *Core*. Essa camada é responsável por casar padrões de acordo com um conjunto de variáveis e condições. O modelo *Core* pode ser diretamente implementado ou utilizado apenas como referencia da semântica da *Relação*. A Relação é facilmente transformada em *Core* utilizando a linguagem de transformação.

A parte imperativa é composta pela camada de *Mapeamento Operacional* e pela *Caixa Preta*. A camada *Mapeamento Operacional* é uma extensão das duas camadas declarativas e possui recursos normalmente encontrados em linguagens imperativas, como *loops* e condições. A outra camada é responsável por juntar facilidades expressas em outras linguagens como XSLT (*Extensible Stylesheet Language Transformations*) e integrar biblioteca que não são QVT (OMG 2004).

Neste trabalho, foi utilizada apenas a parte declarativa e, principalmente, a camada de Relação.

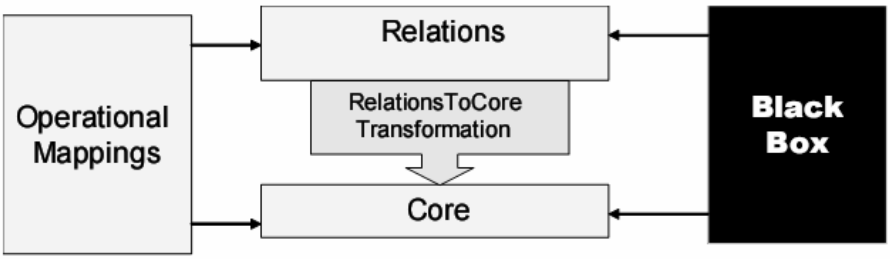

Figura 1 – Relacionamento entre os metamodelos do QVT. Fonte: OMG 2004

4.2 Camada de Relação

Nesta camada, uma transformação é sempre especificada com um conjunto de relações, que devem ser verdadeiras para que a transformação seja efetuada corretamente. Para a transformação, sempre existe um modelo de origem (fonte) e um modelo de destino (alvo). Este modelo de destino pode conter um metamodelo que deve ser respeitado. Uma *relação* é composta de um ou mais domínios e um par de condições, expressas nas clausula *when* e *where*.

Um domínio corresponde aos modelos utilizados para a transformação. Existem duas constantes usadas antes da declaração do domínio: *enforce* e *checkonly*. Normalmente, essa constante é utilizada no domínio resultante. A constante *enforce* tem como objetivo fazer com que o resultado da relação seja positivo, mesmo se para isso seja necessário inserir e alterar o modelo. A constante *checkonly* só irá verificar se a relação está válida ou não. Um exemplo de uma relação geral é mostrado na Figura 2.

A cláusula *when* especifica condições que a relação deve ter. Já a cláusula *where* especifica uma condição que deve ser satisfeita por todos os elementos participantes da relação.

Existem duas formas de relações: nível *top* e não nível *top*. Uma relação com a constante *top* determina que essa relação seja sempre executada, contrário das relações sem essa constante. Normalmente, relações sem essa constante são utilizadas apenas como condição para outra relação.

A equivalência dos padrões pode estar associada com um domínio e isso é chamado de expressões de *template* de objeto (*object template expressions*) (OMG 2004).

```
Relation ClasseToTabela {
  Domain uml c:Classe { namespace = p:Pacote{}, tipo =
         'Persistent', nome = cn
  }
  Domain rdbms t:Table { schema = s: Schema{},
          Nome = cn, column = cl:Column {nome = cn + '
tid'}
         primaryKey = k:PrimaryKey{ nome = cn + '
pk',column = cl}
  }
When { PacoteToSchema(p,s)}
Where { AtributoToColuna (c,t)}
}
```

Figura 2 – Relação de explicação de *object template expressions*

No exemplo anterior, existe um expressão *template* associado com o domínio "*uml*". A correspondência dos padrões irá vincular todas as variáveis da expressão, "c", "p" e "cn", começando pelo variável principal do domínio, "c". A variável "p" já será vinculado através da condição *when.* A correspondência continua agora procurando todos os elementos do tipo Classe to domino que tenham o atributo *tipo= 'Persistent'*. As próximas comparações são com variáveis que estão aninhadas, como é o exemplo do "*namespace*". As funcionalidades de comparar variáveis aninhadas e comparar variáveis que estão dentro das variáveis aninhadas, aumentam ainda mais a complexidade das relações. A cláusula *where* deve possuir ao menos uma linha correspondente do domínio de "*rdbms*" que satisfaça a condição. Caso não exista essa condição, a *Classe* encontrada não é transformada em uma *Tabela* (OMG 2004). Esse *template* permite a criação de elementos no modelo de destino, já que ele está com a constante *enforce.* Todas as características mostradas no segundo domínio, refletem como esse novo objeto criado deve ser estruturado.

Uma transformação também tem um efeito sobre o rastreamento inverso ("*backtraking*"). Se parte das relações falhar, a semântica força "*backtraking*" a ocorrer. Isso acontece porque em diversos pontos da transformação, escolhas podem decidir num caminho que pode resultar em uma falha na transformação subseqüente. Se ocorrer, o rastreamento inverso irá para o último *ponto de escolha*, reiniciando o sistema para o estado anterior, escolhendo um valor diferente do previamente escolhido, continuando assim

a execução. O rastreamento inverso pode ocorrer em um número arbitrário de níveis. Essa técnica permite ao programa encontra um caminho sem falhas sem que haja muita interferência do usuário.

Para aumentar a complexidade das relações é possível utilizar OCL (*Object Constraint Language*), que possibilita a construção de *queries* para a avaliação mais especifica de uma determinada característica.

5 Exemplo de Transformação

5.1 Considerações Iniciais

Esta seção tem como objetivo mostrar um exemplo de transformação entre modelos. A transformação terá como entrada, além deste *script* QVT, os respectivos metamodelos do Diagrama de Atividades e da Rede de *Petri*, além do Diagrama de Atividades que deverá ser transformado. A saída será o Diagrama de Atividades expresso em uma Rede de *Petri*.

Assim como os dois metamodelos, o Diagrama de Atividade utilizando como entrada foi modelado utilizando o *software Eclipse Modeling Framework*. Após essa etapa, foi realizado o mapeamento entre os elementos de cada metamodelo. Foi através desse mapeamento que descrevemos o *script* QVT, que detalharemos no decorrer desta seção. O processo final, descrito na seção 5.3 , contém todos esses elementos, que foram colocados no programa *MediniQVT (*IKV++ *Tecnologies*) para fazer a geração do modelo expresso em uma Rede de *Petri*. Abaixo apresentamos o diagrama de atividade usado como exemplo para a transformação.

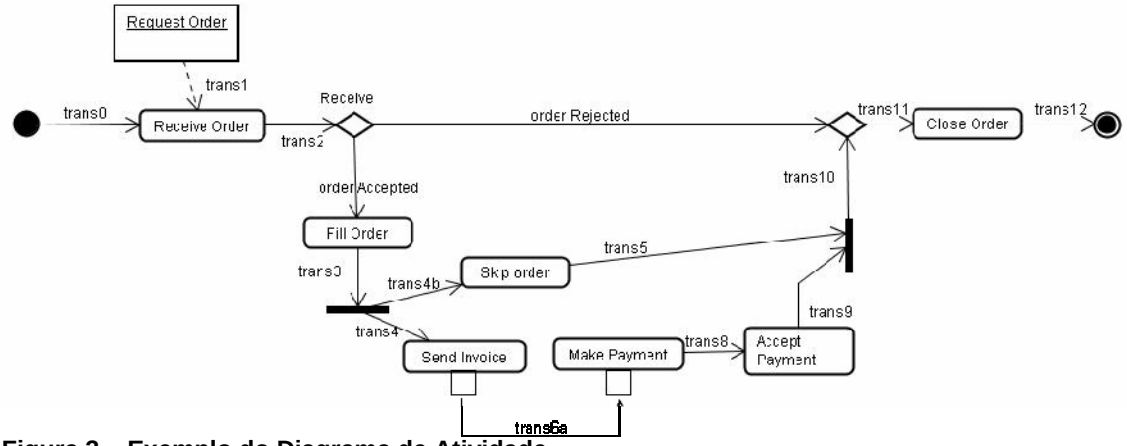

Figura 3 – Exemplo do Diagrama de Atividade

5.2 Construindo o QVT

Para o desenvolvimento do *script* QVT, é necessário primeiramente fazer a relação entre os elementos dos dois metamodelos. Essa relação será descrita na primeira parte dessa seção. Com essas relações desenvolvidas, é possível fazer o *script* QVT que será feito na segunda parte dessa seção.

5.2.1 Mapeamentos dos elementos

O elemento Ação no Diagrama de Atividades representa uma especificação de um comportamento que tem diversos *inputs* que serão transformados em um conjunto de *outputs*. Na Rede de *Petri* o elemento transição é o componente que representa a ação e que pode alterar o estado do sistema. Por possuírem a mesma funcionalidade o elemento Ação é transformado no elemento transição (MAQBOOL,2005).

Como o elemento Atividade Inicial do Diagrama de Atividade apenas passa o fluxo para o próximo elemento, ele será representado como um elemento Lugar sem nenhum marca, pois não demonstra nenhuma mudança de estado (MAQBOOL,2005).

O elemento Decisão direciona o fluxo em diversas direções que são determinadas de acordo com uma Ação anterior. Esse elemento é mapeado como um Lugar sem Marca na Rede de *Petri*. O elemento União é mapeado com um elemento Lugar, assim como o elemento Decisão.

O elemento Separação tem como objetivo duplicar o fluxo e com isso representa uma ação. Logo ele será mapeado como uma Transição na Rede de *Petri*, que tem a mesma função de produzir uma ação. Como o elemento de Junção tem como objetivo unir o elemento Separação, ele também representa uma ação que deve ser representado com uma Transição.

Como já mencionado, os elementos Final de Fluxo e Final de Atividade finalizam o fluxo de *tokens*. Logo, na Rede de *Petri* eles são mapeados para um lugar que não tem saída, chamado de *Sink*. O conceito de finalização de *tokens* não existe em uma Rede de *Petri* e por isso ele não será mapeado na transformação.

O último elemento é o Objeto, ele representa no diagrama de atividade *inputs* e *outputs* para as Ações. Na Rede de *Petri* isso é mapeado como um Lugar com uma Marca que representa a mudança de estado do Objeto em questão. Existe um tratamento diferenciado para o elemento Pino, que é filho do elemento Objeto. Os Pinos podem ser transformados de três diferentes formas: quando tem a relação entre um Pino de Saída e um Pino de Entrada, quando tem apenas um Pino de Saída e quando tem apenas um Pino de Entrada. Na primeira forma os dois elementos são transformados em um elemento Lugar com duas Marcas, uma para cada Pino. Na segunda e terceira forma, o Pino é transformado em um Lugar com uma Marca e o elemento Ação que possui o Pino é ligado com o próximo elemento ou com o elemento anterior do diagrama.

A transformação é feita para cada elemento, mas como existe um seqüenciamento entre os elementos ocorrem alguns problemas. Esses problemas são unicamente causados pela alta complexidade do Diagrama de Atividade e pela simplicidade da Rede de *Petri*. Na Rede de *Petri* existem apenas ligações entre o elemento Transição e o elemento Lugar,. Com isso, quando existe uma seqüência do mesmo elemento, como uma Decisão seguida de outra Decisão, no momento da transformação existe a ligação entre o mesmo elemento na Rede de *Petri*. Por causa deste problema, é necessário a adição de alguns elementos para o modelo resultante estar de acordo com o metamodelo da Rede de *Petri*. Após a correlação entre os elementos dos dois metamodelos e os casos que necessitam adição de elementos será desenvolvido um *script* QVT para traduzir essas relações.

5.2.2 Construção do *script* QVT

Como visto anteriormente, o QVT é constituído de relações entre dois domínios, um de cada metamodelo. Todas as relações utilizam o domínio do Diagrama de Atividade com a constante *checkonly* e o domínio da Rede de *Petri* com a constante *enforce*.

A primeira relação feita foi entre uma Atividade e uma Rede de *Petri*. Esses dois elementos são os nós iniciais dos modelos. Para esses elementos, foi feita uma relação que consiste na criação de um elemento `Rede` na Rede de *Petri* para todos os elementos *ativ* do Diagrama de Atividade, esta relação está descrita na Figura 4.

```
top relation AtividadeToRedePetri {
          pn : String;
          checkonly domain ativ a :Atividade::ativ{
            ativNome= pn
          };

          enforce domain pet p : petri::RedePetri{
            nome = pn
          };
}
```

Figura 4 – Relação Atividade to Rede de *Petri*

É importante mencionar as mudanças que foram realizadas no metamodelo. A primeira modificação feita, foi apenas para não ficar repetindo o atributo nome em todos os elementos, logo foi criado um elemento chamado *nó básico* que tem um atributo nome e um relacionamento com o nó raiz. Todos os outros elementos do metamodelo herdam desse elemento. Essa modificação foi feita nos dois metamodelos. O nó raiz no Diagrama de Atividade é a Atividade e na Rede de *Petri* é o elemento Rede.

Outra modificação feita, foi com o intuito de unir os elementos do Diagrama de Atividade que se transformam no mesmo elemento da Rede de *Petri*. Os elementos modificados são os elementos que herdam `Controle`. Com a correlação feita anteriormente, foi visto que alguns elementos que herdam `Controle` se transformam em uma Transição e outros em um Lugar. Por isso foram criados dois elementos, *Controlet* e *Controlel*.

Foram feitas duas relações para fazer a correspondência de todos os elementos `Controle`. Nas duas relações foi utilizado o mesmo princípio. Para um elemento do domínio do Diagrama de Atividade, é criado um elemento na Rede de *Petri*, onde o nome do elemento criado será o mesmo do elemento no outro domínio. O elemento do Diagrama de Atividade tem relação com um elemento atividade. A cláusula *when* garante que o elemento inserido na Rede de *Petri* tenha a relação com um elemento rede que seja equivalente ao elemento atividade que o outro elemento está relacionado. As duas figuras, Figura 5 e Figura 6, mostram as duas relações para os elementos `Controle`.

```
top relation ControletoTransicao {

          ju : String;

          checkonly domain ativ d : Atividade::controlet{
                 atividade = a : Atividade::ativ{},
                  nome = ju
          };

          enforce domain pet t : petri::Transicao{
            rede = p : petri::RedePetri{},
            nome = ju
          };
          when
          {
                 AtividadeToRedePetri(a,p);
          }
        }
```

Figura 5 – Relação Controle to Transição

```
top relation ControleToLugar {

          ju : String;

          checkonly domain ativ d : Atividade::controlel{
                  atividade = a : Atividade::ativ{},
                  nome = ju
          };

          enforce domain pet t : petri::Lugar{
            rede = p : petri::RedePetri{},
            nome = ju
          };
          when
          {
                 AtividadeToRedePetri(a,p);
          }
        }
```

Figura 6 – Relação Controle to Lugar

Todas as relações criadas utilizam o mesmo raciocínio dessas duas relações. As relações podem ter outras expressões inseridas que serão explicadas ao longo desta seção.

A próxima relação feita foi para o elemento `Objeto`. Para todos os `Objetos` encontrados no domínio do Diagrama de Atividade, eles são transformados em um Lugar com uma `Marca` na Rede de *Petri*. A complicação desta relação é que o elemento `Pino` também é um tipo de `Objeto`. O elemento `Pino` deve ter um tratamento diferente do dado ao `Objeto`. Logo, para isso foi necessário fazer uma relação que verifica que o `Objeto` não pode ser um `Pino`. A chamada desta relação é feita na clausula *when* da relação principal. As duas relações que fazem a transformação estão na Figura 7 e na Figura 8.

```
top relation ObjetotoLugar {

  pn : String;
  checkonly domain ativ b:Atividade::objeto{
    atividade = a : Atividade::ativ{},
    nome = pn
  };

  enforce domain pet t : petri::Lugar{
     marca = s: petri::Marca{},
     rede = p : petri::RedePetri{},
     nome = pn
  };
  when
  {
  AtividadeToRedePetri(a,p);
  not PinoToLugar(b);
  }
}
```

Figura 7 – Relação Principal de Objeto to Lugar

```
top relation PinoToLugar{
  pn : String;
  checkonly domain ativ
b:Atividade::pino{
    atividade = a : Atividade::ativ{},
    nome = pn
  };
}
```

Figura 8 – Relação Secundária de Objeto to Lugar

Outro elemento que sofreu transformação foi a `Ação`. A relação é similar com a relação de `Controle` mostrado anteriormente, só mudando os elemento de origem e o elemento de destino.

O último elemento primário que foi transformado foi o `Pino`. Como visto anteriormente, o `Pino` tem três formas separadas de transformação. Todas as transições entre os elementos transformados foram feitas em uma relação só, que será mostrada no final desta seção. A primeira relação que iremos mostrar é uma `transição` que liga um `Pino de saída` em um `Pino de Entrada`. Para a transformação do `Pino` foi utilizada uma relação que transforma uma `Transição` que tem como origem um `Pino de saída` e como destino um `Pino de entrada` em um lugar com duas marcas. A transformação da ligação dos elementos será mostrada a seguir. A Figura 9 está descrita a relação de transformação.

```
top relation PinoToLugarMarca {

  checkonly domain ativ d : Atividade::transicao{
  atividade = a : Atividade::ativ{},
      origem = b: Atividade::pino_saida{},
      destino = c: Atividade::pino_entrada{}
  };

  enforce domain pet t : petri::Lugar{
    marca = s: petri::Marca{},
    marca = u: petri::Marca{},
    rede = p : petri::RedePetri{},
    nome = 'PINO'
  };
  when
  {
  AtividadeToRedePetri(a,p);
  }
}
```

Figura 9 – Relação Pino Saída e Pino de Entrada com Lugar

A outra forma de transformação é quando tem apenas o `Pino de Saída`. Para resolver esse relacionamento foram necessárias quatro relações: uma que transforma o `Pino de Saída` em um lugar com uma marca, outra que garante que não tenha conflito com a primeira forma de transformação e outras duas para criar as duas ligações que a ação que contem o `Pino de Saída` terá.

A primeira relação tem apenas diferença na cláusula *where.* Nessa cláusula é utilizado o operador lógico *not* com a segunda relação. A segunda relação garante que o `Pino de Saída` não tem relação com nenhum `Pino de Entrada`, para que não sejam sobrepostos as transformações.

A Figura 10 tem a relação que transforma o `Pino de Saída` e a Figura 11 mostra relação que evita conflitos com a primeira transformação de `Pino`.

```
top relation PinoSaidaToLugarMarca {

  ju : String;
  checkonly domain ativ d : Atividade::pino_saida{
  atividade = a : Atividade::ativ{},
  nome = ju
  };

  enforce domain pet t : petri::Lugar{
    marca = s: petri::Marca{},
    rede = p : petri::RedePetri{},
    nome = ju
  };
  when
  {
  AtividadeToRedePetri(a,p);
    not PinoSaida(d);
  }
}
```

Figura 10 – Relação do Pino de Saída para Lugar

```
top relation PinoSaida{
  pn : String;
  checkonly domain ativ
b:Atividade::pino_saida{
    atividade = a : Atividade::ativ{},
    origemTransacao = c:
Atividade::transicao{
    destino = d:
Atividade::pino_entrada{}
    },
    nome = pn
  };
}
```

Figura 11 - Relação para evitar conflitos com a outra transformação

Para simplificar a explicação, o elemento `Ação` que possui o `Pino` será denominado portador e o elemento para qual o `Pino` aponta será chamado de destinatário. As outras duas relações feitas para a transformação do elemento `Pino` são: uma para fazer a ligação entre o portador e o destinatário e outra para fazer ligação entre portador e o elemento que o `Pino` se transformou. Na primeira relação descrita acima, a principal inovação é como é feita a recuperação dos dois elementos participando da ligação, o elemento portador e o destinatário. Para identificar o elemento portador, foi necessário fazer o acesso através da transição de origem e nesse último elemento acessar a origem. O mesmo foi feito para identificar o elemento destinatário. Na Figura 12 e Figura 13 mostra as duas relações explicadas anteriormente.

```
top relation TransicaoToLugarViaPino {

  ju : String;
  checkonly domain ativ d : Atividade::pino{
  atividade = a : Atividade::ativ{},
  destinoTransacao = b :Atividade::transicao{
  origem = c: Atividade::noatividade{}
  },
  origemTransacao = e :Atividade::transicao{
  destino = f : Atividade::noatividade[]
  },
  nome = ju
  };

  enforce domain pet t : petri::OutputArc{
    rede = p : petri::RedePetri{},
lugar = r :petri::Lugar{},
trans = s :petri::Transicao{},
nome = ju
  };
  when
  {
  AtividadeToRedePetri(a,p);
  AcaoToTransicao(c,s)or ControleToTransicao(c,s);
  ControleToLugar(f,r);
  }
}
```

Figura 12 – Relação de ligação entre o elemento que possui o Pino e quem o pino aponta

```
top relation TransicaoToLugarViaPino2 {

  ju : String;
  checkonly domain ativ d : Atividade::pino{
  atividade = a : Atividade::ativ{},
  destinoTransacao = b :Atividade::transicao{
  origem = c: Atividade::noatividade{}
  },
  origemTransacao = e :Atividade::transicao{
  origem = f : Atividade::noatividade{}
  },
  nome = ju
  };

  enforce domain pet t : petri::OutputArc{
    rede = p : petri::RedePetri{},
lugar = r :petri::Lugar{},
trans = s :petri::Transicao{},
nome = ju
  };
  when
  {
  AtividadeToRedePetri(a,p);
  AcaoTcTransicao(c,s)or ControletoTransicao(c,s);
  PinoSaidaToLugarMarca(f,r);
  }
}
```

Figura 13 – Relação de ligação entre o Pino e o elemento que possui o Pino

A última forma de transformação é quando tem apenas o `Pino de Entrada`. Para resolver esse relacionamento foram necessárias quatro relações, parecida com as relações feitas para a última forma de transformação de `Pino`. A única mudança que nas duas primeiras relações, as relações de transformação do `Pino` e garantia de não ter conflito com a primeira forma de transformação, foi o elemento do domínio do Diagrama de Atividades que foi utilizado, neste caso `Pino de Entrada`. Nas outras duas relações, a diferença foi qual era o tipo de elemento que será transformado, que neste caso é o Arco de *Input*..

A próxima relação feita foi para transformar a ligação entre os elementos, no caso do Diagrama de Atividade o elemento Transição e na Rede de *Petri* os elementos Arco *Input* e o Arco *Output*. O Arco de *Input*, sempre tem origem em um `Lugar` e destino em uma Transição. Com o Arco de *Output*, é exatamente o contrário, origem é em uma Transição e o destino em um `Lugar`. Por causa dessa característica, foi feito duas relações, uma para todas as Transições que iriam virar Arcos *Input* e outra para os que iriam virar Arcos de *Output*.

Na relação de Arcos de *Input*, foi feita uma relação genérica, que tem a origem e o destino como um `Nó Atividade`, que é pai de todos os elementos com exceção da `Transição`. No domínio da Rede de *Petri* dessa relação, sempre será criado um elemento Arco de *Input* e sempre serão referenciados um lugar e uma transição. Foram analisados todos os elementos que poderiam ser origem e todos que podiam ser destino e foi utilizada a cláusula *where* para garantir essa analise. Essa relação foi desenvolvida para garantir que apenas essa relação resolvesse todas as transformações para Arco de *Input*. A Figura 14 demonstra relação Arco de *Input* e tem como objetivo resolver todos os problemas mencionados.

```
top relation ArcosInput{

nom : String;
checkonly domain ativ f :Atividade::transicao
{
atividade = a : Atividade::ativ{},
origem = e : Atividade::noatividade{},
destino = h :Atividade::noatividade{},
nome = nom
};

enforce domain pet g : petri::InputArc
{
rede = p : petri::RedePetri{},
lugar = r :petri::Lugar{},
trans = s :petri::Transicao{},
nome = nom
};
when
{
AtividadeToRedePetri(a,p);
ControleToLugar(e,r) or ObjetotoLugar(e,r) or Pino(e,r);
ControletoTransicao(h,s) or AcaoToTransicao(h,s);
}

}
```

Figura 14 – Relação dos Arcos Input

O mesmo foi feito para o Arco de Output. Foi feito uma relação genérica e na cláusula *when* adicionando os elementos que poderiam ser origem e os que poderiam ser destino. A relação Arco de Output está descrita abaixo.

```
top relation ArcosOutput{

nom : String;
checkonly domain ativ f :Atividade::transicao
{
atividade = a : Atividade::ativ{},
origem = e : Atividade::noatividade{},
destino = h :Atividade::noatividade{},
nome = nom
};

enforce domain pet g : petri::OutputArc
{
rede = p : petri::RedePetri{},
lugar = r :petri::Lugar{},
trans = s :petri::Transicao{},
nome = nom
};
when
{
AtividadeToRedePetri(a,p);
AcaoToTransicao(e,s) or ControletoTransicao(e,s);
ControleToLugar(h,r) or ObjetotoLugar(h,r) or Pino(h,r);
}
}
```

Figura 15 – Relação de Arcos de Output

Como mostrado no inicio desta seção, existem algumas simplificações necessárias. Todas elas foram resolvidas dentro do QVT.

As simplificações foram divididas em três grandes grupos. Para os três grupos, foi necessário fazer três relações. A primeira relação é responsável por criar o elemento *Dummy*, dependendo dos elementos da relação. As duas outras relações são para fazer a ligação entre os elementos inicias da relação e o elemento *Dummy* criado.

O primeiro grupo é dos elementos do Diagrama de Atividade que se transformam no elemento Lugar da Rede de *Petri*, com exceção ao elemento Pino. Como dito anteriormente, existem 4 conjuntos de elementos que sofreram simplificação nesse grupo. São eles: Decisão com Decisão, União para União, União para Decisão, Decisão para União. Cada um desses conjuntos terá a primeira relação, com a criação do elemento *Dummy*. Todas as quatro primeiras relações são parecidas, trocando apenas qual elemento do domínio do Diagrama de Atividade será utilizado como origem e qual será utilizado como destino. No exemplo abaixo, está descrita a relação entre `Decisão e União`.

```
relation SimplificacaoDecisaoToUniao{

                nom : String;
                checkonly domain ativ f :Atividade::transicao
                {
                        atividade = a : Atividade::ativ{},
                        origem = e : Atividade::decisao{},
                        destino = h :Atividade::uniao{},
                        nome = nom
                };

                enforce domain pet g : petri::Transicao
                {
                        rede = p : petri::RedePetri{},
                        nome = nom + 'DUMMY'
                };
                when
                {
                        AtividadeToRedePetri(a,p);
                }

        }
```

Figura 16 - Relação de Simplificação de Decisão para União

Já a segunda e a terceira relação são reutilizáveis para esses quatro conjuntos. Essa reutilização é feita através do domínio do Diagrama de Atividade com o atributo origem e o destino do elemento Transição através da utilização do elemento `Controlel`, que é pai de todos os elementos deste grupo. A restrição de quais elementos podem ser usados na relação é inserido na cláusula *when*. O destino dessa ligação é sempre um lugar, logo na cláusula *when* a relação primária de transformação entre um elemento `Controle` e um lugar. A origem da ligação é sempre um elemento que se transforma em uma Transição, logo foram utilizadas todas as primeiras relações dos 4 conjuntos para garantir a correlação entre os elementos. A diferença para as outras relações é o tipo de elemento que é criado, neste caso Arco de *Input*, e acontece exatamente o contrário da primeira relação. A origem tem um lugar associado e o destino uma Transição, gerada a partir da primeira relação de simplificação. As duas relações estão mostradas a seguir.

```
top relation SimplificacaoContr2{

                nom : String;
                checkonly domain ativ f :Atividade::transicao
                {
                        atividade = a : Atividade::ativ{},
                        origem = e : Atividade::controlel{},
                        destino = h :Atividade::controlel{},
                        nome = nom
                };

                enforce domain pet g : petri::OutputArc
                {
                        rede = p : petri::RedePetri{},
                        lugar = s :petri::Lugar {},
                        trans = r:petri::Transicao{},
                        nome = nom

                };
                when
                {
                        AtividadeToRedePetri(a,p);
                        SimplificacaoDecisaoToUniao(f,r) or
                                SimplificacaoDecisaoToDesicao(f,r) or
SimplificacaoUniaoToDecisao(f,r) or
SimplificacaoUniaoToUniao(f,r) or
SimplificacaoDecisaoToFim(f,r);
                        ControleToLugar(h,s);
                }

        }
```

Figura 17 – Primeira Relação de Simplificação do primeiro grupo

```
top relation SimplificacaoContr3{

                nom : String;
                checkonly domain ativ f :Atividade::transicao
                {
                        atividade = a : Atividade::ativ{},
                        origem = e : Atividade::controlel{},
                        destino = h :Atividade::controlel{},
                        nome = nom
                };

                enforce domain pet g : petri::InputArc
                {
                        rede = p : petri::RedePetri{},
                        lugar = s :petri::Lugar {},
                        trans = r:petri::Transicao{},
                        nome = nom
                };
                when
                {
                        AtividadeToRedePetri(a,p);
                        SimplificacaoDecisaoToUniao(f,r) or
SimplificacaoDecisaoToDesicao(f,r) or
SimplificacaoUniaoToDecisao(f,r) or
SimplificacaoUniaoToUniao(f,r) or
SimplificacaoDecisaoToFim(f,r);
                        ControleToLugar(e,s);
                }

        }
```

Figura 18 - Segunda Relação de Simplificação do primeiro grupo

O segundo grupo é dos elementos do Diagrama de Atividade que se transformam no elemento Transição da Rede de *Petri*. Como dito anteriormente, existem 8 conjuntos de elementos que sofreram simplificações nesse grupo, que são: `Ação` com `Ação`, `Ação` com `Junção`, `Ação` com `Separação`, `Junção` com `Junção`, `Junção` com `Ação`, `Junção` com `Separação`, `Separação` com `Separação`, `Separação` com `Ação`, `Separação` com `Junção`. A forma de desenvolvimento das relações é igual ao grupo anterior. A primeira relação existe para cada conjunto de elementos e a segunda e terceira relação é única para todo o grupo. Os elementos `Separação` e `Junção` estão classificados como `Controlet`, com visto anteriormente. Por causa dessa classificação, passaram a existir apenas as 4 primeiras relações. A diferença entre as relações neste caso é qual o tipo de elemento do domínio do Diagrama de Atividades que é igual à origem e ao destino. A relação de uma `Ação` para uma `Ação` está descrita a seguir.

```
top relation SimplificacaoAcaotoAcao{

                nom : String;
                checkonly domain ativ f :Atividade::transicao
                {
                        atividade = a : Atividade::ativ{},
                        origem = e : Atividade::acao{},
                        destino = h :Atividade::acao{},
                        nome = nom
                };

                enforce domain pet g : petri::Lugar
                {
                        rede = p : petri::RedePetri{},
                        nome = nom + 'DUMMY'
                };
                when
                {
                        AtividadeToRedePetri(a,p);
                }

        }
```

Figura 19 – Relação de Simplificação de Ação to Ação

A segunda e a terceira relação são parecidas com a do primeiro grupo, com a diferença apenas em quais relações são utilizadas na clausula *where*. Neste caso são as primeiras relações criadas para este grupo. Outra diferença é que além dessas relações, é utilizada a relação primária *AçãoToTransicao* e *ControletToTransicao* para garantir que a transformação para o elemento Transição. As duas relações estão expressas a seguir.

```
top relation Simplificacao2{

                nom : String;
                checkonly domain ativ f :Atividade::transicao
                {
                        atividade = a : Atividade::ativ{},
                        origem = e : Atividade::noatividade{},
                        destino = h :Atividade::noatividade{},
                        nome = nom
                };

                enforce domain pet g : petri::OutputArc
                {
                        rede = p : petri::RedePetri{},
                        lugar = s :petri::Lugar {},
                        trans = r:petri::Transicao{},
                        nome = nom

                };
                when
                {
                        AtividadeToRedePetri(a,p);
                        SimplificacaoAcaotoAcao(f,s) or
                                SimplificacaoAcaoToControlet(f,s) or
SimplificacaoControletToControlet(f,s) or
SimplificacaoControletToAcao(f,s);
                        AcaoToTransicao(e,r) or ControletoTransicao(e,r);
                }

        }
```

Figura 20 – Segunda relação de simplificação do segundo grupo

```
        top relation Simplificacao3{

                nom : String;
                checkonly domain ativ f :Atividade::transicao
                {
                        atividade = a : Atividade::ativ{},
                        origem = e : Atividade::noatividade{},
                        destino = h :Atividade::noatividade{},
                        nome = nom
                };

                enforce domain pet g : petri::InputArc
                {
                        rede = p : petri::RedePetri{},
                        lugar = s :petri::Lugar {},
                        trans = r:petri::Transicao{},
                        nome = nom
                };
                when
                {
                        AtividadeToRedePetri(a,p);
                        SimplificacaoAcaotoAcao(f,s) or
SimplificacaoAcaoToControlet(f,s) or
SimplificacaoControletToControlet(f,s) or
SimplificacaoControletToAcao(f,s);
                        AcaoToTransicao(h,r) or ControletoTransicao(h,r);
                }

        }
```

Figura 21 - Terceira relação de simplificação do segundo grupo

A ultima simplificação realizada foi para o elemento `Pino`. Essa simplificação teve que ser separada já que o acesso aos tipos de elementos que participam das transformações é feito dentro de uma variável aninhada. Foram feitas duas simplificações, uma quando tem apenas o elemento `Pino de Saída` e outra quando tem apenas o elemento `Pino de Entrada`. A estrutura das duas simplificações está igual ao das simplificações já explicadas.

Para simplificar a explicação, o elemento que possui o `Pino` será denominado portador e o elemento para qual o elemento `Pino` aponta será chamado de destinatário. A

primeira relação a seguir descreve a transformação de um `Pino de Saída` que tem como origem o portador, que é um elemento `Ação` e como destino o elemento destinatário, que também é um elemento `Ação`. A segunda relação mostra a ligação entre o elemento portador e o elemento em qual o `Pino` se transformou, e a terceira relação a ligação entre o elemento portador e o elemento destinatário. A diferença dessa simplificação para todas as demonstradas anteriormente, é a forma que o elemento origem e o elemento destino são recuperados. Para isso, é necessário acessar através da transição que o `Pino` possui o elemento que é origem dessa transição. Esse mecanismo foi utilizado para todas as relações descritas a seguir.

```
top relation TransicaoToTransicaoViaPino {

          ju : String;
          checkonly domain ativ d : Atividade::pino_saida{
                         atividade = a : Atividade::ativ{},
                         destinoTransacao = b :Atividade::transicao{
                                  origem = c: Atividade::acao{}
                         },
                         origemTransacao = e :Atividade::transicao{
                                  destino = f : Atividade::acao{}
                         },
                         nome = ju
          };

          enforce domain pet t : petri::Lugar{
            rede = p : petri::RedePetri{},
                 nome = ju + 'DUMMY'
          };
          when
          {
                 AtividadeToRedePetri(a,p);
          }
        }
```

Figura 22 – Simplificação Pino de saída I

```
        top relation TransicaoToTransicaoViaPino2 {

          ju : String;
          checkonly domain ativ d : Atividade::pino{
                         atividade = a : Atividade::ativ{},
                         destinoTransacao = b :Atividade::transicao{
                                  origem = c: Atividade::noatividade{}
                         },
                         origemTransacao = e :Atividade::transicao{
                                  destino = f : Atividade::noatividade{}
                         },
                         nome = ju
          };

          enforce domain pet t : petri::OutputArc{
            rede = p : petri::RedePetri{},
                 lugar = r :petri::Lugar{},
                 trans = s :petri::Transicao{},
                 nome = ju
          };
          when
          {
                 AtividadeToRedePetri(a,p);
                 AcaoToTransicao(c,s);
                 TransicaoToTransicaoViaPino(d,r);
          }
        }
```

Figura 239 – Simplificação Pino de saída II

```
top relation TransicaoTcTransicaoViaPino3 {

  ju : String;
  checkonly domain ativ d : Atividade::pino{
                atividade = a : Atividade::ativ[],
                destinoTransacao = b :Atividade::transicao{
                        origem = c: Atividade::noatividade{}
                },
                origemTransacao = e :Atividade::transicao{
                        destino = f : Atividade::noatividade{}
                },
                nome = ju
  };

  enforce domain pet t : petri::InputArc{
    rede = p : petri::RedePetri{},
         lugar = r :petri::Lugar{},
         trans = s :petri::Transicao{},
         nome = ju
  };
  when
  {
         AtividadeTcRedePetri(a,p);
         AcaoToTransicao(f,s);
         TransicaoTcTransicaoViaPino(d,r);
  }
}
```

Figura 60 – Simplificação Pino de saída II

A simplificação realizada para o elemento `Pino de Entrada` é similar a simplificação do elemento `Pino de Saída`. As únicas diferenças são o elemento do domínio na primeira relação, que neste caso é o `Pino de Entrada` e qual relação primária será utilizado para as outras duas relações, neste caso utilizou-se a primeira relação de simplificação do `Pino de Entrada`.

5.3 Transformação utilizando MediniQVT

Para a transformação de modelos foi utilizado o *software* MediniQVT, que foi o único *software* livre encontrado que faz a transformação da camada de Relação do QVT.

A ferramenta *MediniQVT* é também baseada no *Eclipse*, mas é um *software* fechado. No entanto, ele foi o escolhido para este trabalho por utilizar QVT Relação e consegue fazer a partir de qualquer metamodelo a transformação em qualquer outro metamodelo dado com *input*. Esse *software* é desenvolvido *ikv++ Technologies ag* e a versão utilizada do *software* foi 1.0.0.

Para utilizar esse sistema é necessário criar um projeto que tenha uma pasta com os dois metamodelos, dois arquivo *.ecore*, e o exemplo do modelo, um arquivo *.xmi*. Sendo também necessário ter pasta com o *script* QVT desenvolvido e uma pasta de rastreamento.

Um problema encontrado no momento de fazer a transformação, foi que alguns atributos tiverem que ser modificados para que pudesse ser entendida pelo *software*. Os atributos alterados foram: *container*, *containment* e *EOpossite*. Para o lado 1 da relação, os atributos têm os seguintes valores: *container* = *true*, *containment* = *false* e o *EOpossite* irá referenciar o relacionamento onde está a parte N do relacionamento. Já para o lado N do relacionamento, os atributos têm os seguintes valores: *container* = *false, containment* = *true* e *EOpossite* irá referenciar o relacionamento onde está a parte N do relacionamento.

Depois de configurar esses detalhes, é possível executar a transformação e o arquivo *Petri*.xmi é criado. A seguir está o modelo da Rede de *Petri* gerada.

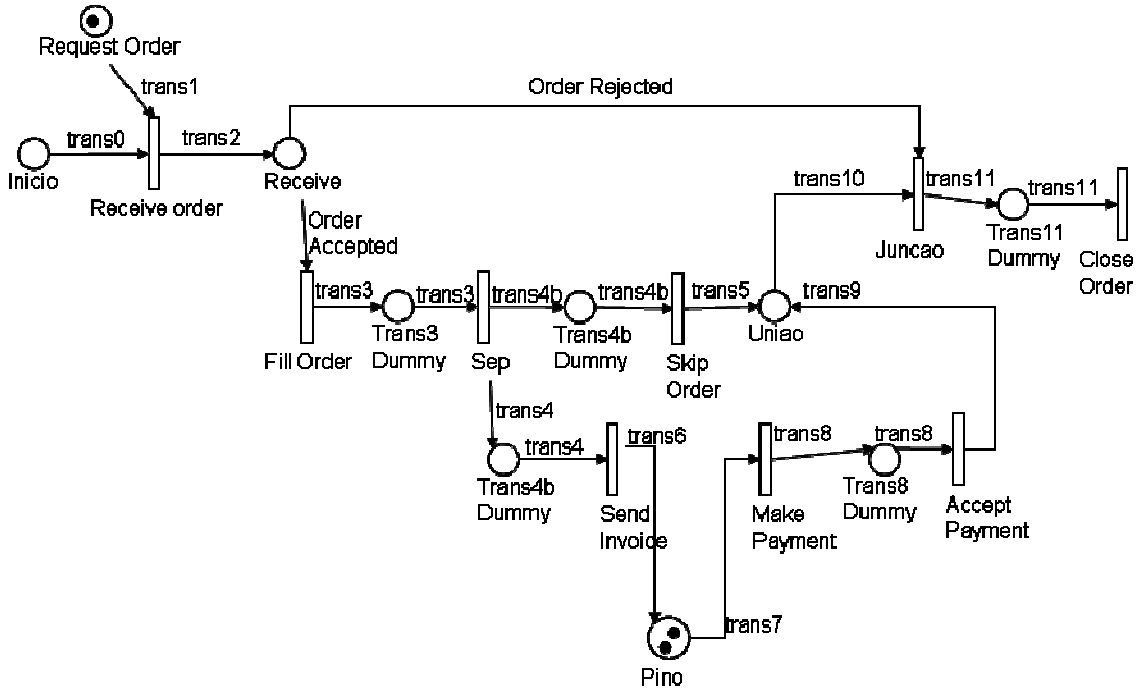

Figura 5 – Rede de *Petri* transformada.

6 Considerações Finais

O objetivo desse trabalho foi realizar um estudo sobre QVT. Para tal, transformou um Diagrama de Atividades da UML 2.0 para uma Rede de *Petri*, utilizando a linguagem QVT padronizada pelo OMG, exemplificando com a transformação de dois modelos que tem o mesmo nível de abstração.

A importância deste estudo está no fato de que os desenvolvedores não precisem utilizar o Diagrama de Atividades para finalidades as quais, ele não foi estruturado para realizar. Além disso, é importante verificar que é possível fazer a transformação entre outros modelos, que podem proporcionar benefícios futuros ainda não estudados e/ou explorados.

No decorrer desse trabalho ocorreram algumas dificuldades na sua execução. Entre elas podemos destacar a complexidade do *Eclipse Modeling Framework* (EMF) e na utilização do *software MediniQVT*.

Com relação ao EMF, destacamos o fato de que quando os arquivos eram gerados, a importação do arquivo de biblioteca e outros arquivos fontes eram feitos de forma incorreta, provocando erros e sendo necessários consertos manuais. Outra dificuldade encontrada foi, ao reabrir um projeto, que já possui o *.edit* e o *.editor* criados, ocorriam erros de incompatibilidade com a versão da biblioteca JRE (*Java Runtime Enviroment*) do *Java*. Nesse caso o procedimento adotado foi o de excluir os dois arquivos e fazer a geração novamente, uma vez que assim superávamos o problema.

Já no *software MediniQVT* utilizado, o problema acontecia quando se executava uma alteração do arquivo *.xmi* que continha o exemplo a ser transformado. Quando havia a alteração do modelo a transformação se perdia e provocando um erro de falta de refe-

rencia do elemento excluído. Uma das soluções possíveis para esse problema era a troca do nome do arquivo que continha o exemplo. Outro problema acontecia ao abrir o arquivo *.xmi* do modelo transformado pela primeira vez, que provocava o erro *"Unable to create this part due to an internal error. Reason for the failure: assertion failed:"*. A solução encontrada foi a de realizar uma segunda tentativa para que o erro não persistisse.

No decorrer dos estudos observamos que o potencial desta proposta foi significativamente superior ao previsto no início do mesmo.

Como sugestão para continuidade deste trabalho a extensão das Redes de *Petri* para que se possa incorporar todos os elementos que fazem parte do Diagrama de Atividades da UML 2.0. Outra melhoria que poderia ser incluída é a criação de uma ferramenta que interprete o XML e gere o desenho do modelo tanto do Diagrama de Atividades quanto da Rede de *Petri*.

7 Referências

BOAVENTURA NETTO, P. O., **Grafos: Teoria, Modelos, Algoritmos,** 4a. Edição, Editora Edgard Blücher Ltda., São Paulo, 2006.

BOOCH, G., **Object Oriented Analysis and Design with Applications**, 2ª Edition, Addison-Wesley, ISBN 0-8053-5340-2, 1994.

BOOCH, G., RUMBAUGH, J. e JACOBSON, I., **The Unified Modeling Language User Guide**, Addison-Wesley, 2a Edition, ISBN: 0321267974, 2005.

HAMMER, M. e CHAMPY J., **Reengenharia Revolucionaria a Empresa**, Editora Campus, 1994.

HEUSER, C. A., **Modelagem Conceitual de Sistemas – Redes de *Petri***, Edição Preliminar, Publicada para a V Escola Brasileiro-Argenina de Informática, 1990.

FOWLER, M., **UML Distilled: A Brief Guide to the Standard object Modeling Language**, 3a Edition, Pearson Education, ISBN: 0321193687, 2004.

GARDNER, T., GRIFFIN, C., KOEHLER, J., *et al.*, 2002, **A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard,** OMG Document: ad/03-08-02.

MAQBOOL, S. **Tranformation of a Core Scenario and Activity Diagrams into *Petri* Net**, Master In Computer Science in University of Ottawa, September 2005

MELLOR, J. S., SCOTT, K., UHL A. e WEISE, D., **MDA Distilled: Principles of Model-Driven Architecture**, Addison-Wesley, ISBN: 0-201-78891-8, 2004.

MEYER, B., Object-Oriented *Software* Construction, Prentice-Hall, 1988.

MURATA, T., ***Petri* Nets: Properties, Analysis and Applications**. Proceedings of the IEEE, 77(4): 541-580. April 1989.

OMG, Object Management Group - **QVT - Query/Views/Transformations**, version 1.8, 2004. Disponivel em: www.omg.org, acessado em: 10/10/2006.

OMG, Object Management Group - **Unified Modeling Language (UML) Superstructure Specification**, version 2.0, 2005. Disponivel em: http://www.omg.org/cgi-bin/doc?formal/05-07-04, Acessada em: 01/2007.

O'NEILL, P. e SOHAL, A., **Business Process Reengineering: A Review of Recent Literature**, Technovation, vol. 19, no. 9, pp. 571-581, 1999.

PETERSON, J. L., ***Petri* Nets. ACM Computing Surveys**, 9(3): 223-252. September 1977.

PETRI, C. A., Kommunikation Mit Automaten. PhD Thesis, Technische Hochschule Darmstadt, 1961.