VERIFICAÇÃO AUTOMATIZADA DE PROGRAMAS ESCRITOS EM SISTEMA DE RACIOCÍNIO PROCEDURAL (PRS) UTILIZANDO REDES DE PETRI COLORIDAS (RPCS)

Ricardo Wagner de Araújo[*]
ricardo@cerescaico.ufrn.br

Adelardo A. D. de Medeiros[†]
adelardo@dca.ufrn.br

[*]Universidade Federal do Rio Grande do Norte - UFRN
Departamento de Ciências Exatas e Aplicadas - DCEA - CERES

[†]Universidade Federal do Rio Grande do Norte - UFRN
Departamento de Engenharia de Computação e Automação - DCA - CT

ABSTRACT

This article presents a metodology used in the development of an automatic converter of Procedural Reasoning Systems (PRS) programs to Coloured Petri Nets (CPNs). The main objectives of this work is the presentation of the algorithms developed for the converter as well as its aspects of implementation.This converter will be used to proceed to the formal verification of programs PRS through the formalism of the Petri Nets analysis methods.

KEYWORDS: Procedural Reasoning System, Coloured Petri Nets, Formal Verification, Algorithm

RESUMO

Este artigo apresenta uma metodologia usada no desenvolvimento de um conversor automático de programas escritos em Sistemas de Raciocínio Procedrual (PRS) para Redes de Petri Coloridas (CPNs). O objetivo principal deste trabalho é a apresentação dos algoritmos desenvolvidos para o conversor assim como seus aspectos de implementação. As redes de Pertri geradas pelo conversor serão utilizadas para proceder à verificação dos programas PRS equivalentes, através do formalismo das mesmas. São também apresentados alguns resultados preliminares dessa abordagem.

PALAVRAS-CHAVE: Sistema de Raciocínio Procedural, Redes de Petri, Verificação Formal, Algoritmos

1 INTRODUÇÃO

Como a tecnologia de sistemas baseados em conhecimento ganhou larga aceitação nas últimas décadas, existe uma necessidade crescente de verificação de tais sistemas com o intuito de melhorar sua confiabilidade e qualidade (Bench-Capon et al., Jun,1993). Os sistemas automatizados para supervisão e diagnósticos são ferramentas essenciais no controle de processos industriais. Freqüentemente, essas tarefas são melhor manipuladas por regras ou procedimentos que usam conhecimento específico da situação, ao invés de técnicas de controle convencionais.

Uma das maiores dificuldades no projeto de tais sistemas é o controle em tempo-real de sua execução. Muitas das técnicas usuais de representação do conhecimento e inferência em Inteligência Artificial não têm um tempo de resposta garantido e portanto não são aplicáveis a tais sistemas.

Entre as possíveis alternativas nessas situações, podemos destacar o raciocínio procedural (Georgeff e Lansky, 1986; Ingrand et al., December 1992). A tecnologia de raciocínio procedural foi desenvolvida inicialmente no *Artificial Intelligence Center of the Stanford Research Institute (Menlo Park, California)* (International, 1994). O raciocínio procedural difere de outras representações usuais do conhecimento porque não preserva algum fluxo de informação (isto é, a seqüência de ações e testes) associado aos aspectos declarativos.

Uma das implementações mais usadas do raciocínio procedural é o PRS (*Procedural Reasoning System*). Ele tem sido usado em aplicações em tempo-real – tais como supervisão e controle de robôs móveis (Ingrand e Despouys, 2001), detecção de falhas em ônibus espaciais (International, 1994) assim como no gerenciamento de redes de comunicação – devido à sua eficiência temporal e alto poder de expressão. Em sistemas complexos, entretanto, às vezes essas propriedades não são suficientes: precisamos de algumas garantias formais concernentes à corretude do sistema supervisório.

PRS não oferece um mecanismo de verificação formal. Este artigo apresenta uma metodologia desenvolvida para a concepção de um conversor automático de PRS para redes de Petri coloridas bem como um pequeno estudo de caso para

ilustrar a abordagem e obter alguns resultados preliminares. O objetivo do conversor é proceder à tradução de programas escritos em PRS para RPC. Assim, será possível fazer a verificação formal de tais programas usando-se os formalismos das redes de Petri.

2 A LINGUAGEM PRS

As características principais do PRS foram melhor abordadas em (Araújo e Medeiros, 2004). Para nossos propósitos, podemos dizer que um agente PRS consiste de :

Biblioteca de rotinas: cada rotina, ou KA (knowledge area), é uma seqüência de ações e/ou testes que pode ser executada para atingir metas ou reagir a situações.

Banco de Dados: contém fatos que representam a visão corrente do mundo vista pelo sistema.

Conjunto de metas correntes: em PRS, as metas descrevem tarefas ou comportamentos desejados. Na lógica usada pelo PRS, a meta para alcançar uma certa condição `C` é escrita como `(! C)`; para testar uma condição, como `(? C)`; para esperar até que uma condição seja verdadeira, como `(^ C)`; para manter `C`, como `(# C)`; para concluir a condição `C` no banco de dado, como `(=> C)` e para retirá-la do banco de dados, como `(~> C)`.

Grafo de intenções: é um conjunto dinâmico de rotinas, estruturadas como um grafo, em que o sistema mantém informações em tempo-real sobre o estado das rotinas escolhidas para execução e suas sub metas postadas.

Cada KA tem um corpo e algumas condições de invocação. Para exemplificar, uma KA que é parte de um sistema de controlador de elevador é mostrada na Figura 1. Esta KA é considerada para possível execução quando uma ordem `IR-PARA` para realizar uma meta é postada (como o campo `INVOCATION` mostra). Ela pode ser selecionada se as condições do campo `CONTEXT` forem satisfeitas. As condições contém variáveis a unificar (`@F`, no caso). Existem dois tipos de variáveis em PRS: as variáveis lógicas `$var` têm o comportamento clássico de variáveis em linguagem de programação (uma vez unificadas, não mudam seu valor), enquanto as variáveis de programa `@var` podem ser re-atribuídas.

O corpo da KA descreve sua execução. A execução inicia no nó `START` e continua seguindo os arcos através da rede. Se mais de um arco deixa um nó, qualquer um deles pode ser atravessado. Para atravessar um arco, o sistema deve ou testar se a meta associada ao arco já foi estabelecida no banco de dados ou lançar uma KA que atinja a meta associada àquele arco. Esta nova meta será incorporada ao grafo de intenções e deverá ser satisfeita antes que a KA corrente possa ser satisfeita. Se o sistema não pode satisfazer nenhuma das metas associadas aos arcos que deixam o nó, a KA inteira falha. Mas quando um nó terminal (nó `END`) é alcançado, a meta é considerada satisfeita e os fatos listados no campo `EFFECTS` são concluídos na base de dados.

3 AS REDES DE PETRI COLORIDAS

As redes de Petri coloridas são largamente conhecidas e podem ser encontradas na literatura (Jensen, 1994). Um exemplo de

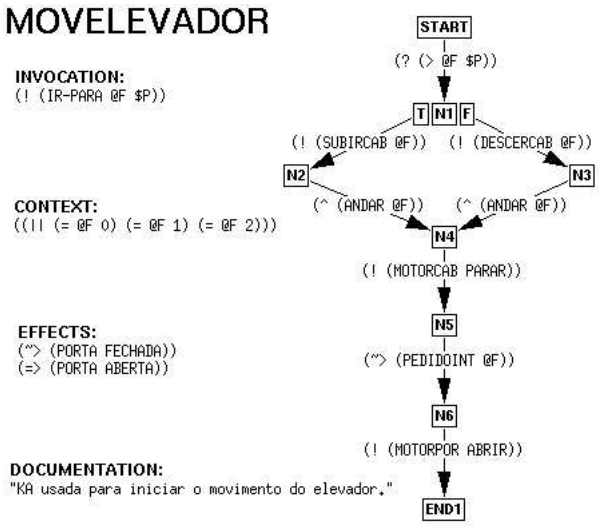

Figura 1: Exemplo de uma KA.

modelagem (Jensen, 1998) de uma transição habilitada de um sistema de alocação de recursos é mostrado na Figura 2.

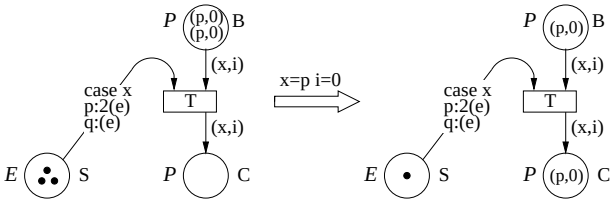

Figura 2: Disparo de uma transição.

Cada lugar (círculos que descrevem os estados do sistema - B, C e S) contém um conjunto de marcas chamadas `fichas` as quais carregam um dado valor e pertencem a um `tipo` (*P* e *E*, no caso). O lugar `B` tem duas fichas, no estado inicial, de cor (p, 0), tipo *P*. O lugar `S` tem três fichas de cor(e), tipo *E*.

Para ser disparada, uma transição (retângulos que descrevem as ações) deve ter fichas suficientes nos seus lugares de entrada, e essas fichas devem ter valores que unifiquem as expressões de arco (setas que descrevem como o estado das redes de Petri mudam quando uma transição dispara). Considere a transição `T`. Ela tem dois arcos. As duas expressões de arco envolvem as variáveis `x` e `i`. Para disparar a transição, temos de unificar essas duas variáveis a valores de seus tipos, de tal maneira que a expressão de cada arco de entrada avalie a um valor de ficha que esteja presente no lugar de entrada correspondente. Nesse caso, `x=p` e `i=0`. O lado direito da Figura 2 mostra o novo estado após a transição `T` disparar.

4 EXEMPLO DE REGRA DE CONVERSÃO PRS-CPN

Apresentamos nesta seção um exemplo de regra de conversão PRS-CPN bem como seu algoritmo e a codificação em Lisp (linguagem de programação usada para implementar o conversor). O teste (? (...)) sobre predicados padrão são representados em CPN como mostra a Figura 3. Para conhecer outras regras de conversão e mais detalhes sobre esta, ver

(Araújo e Medeiros, 2004).

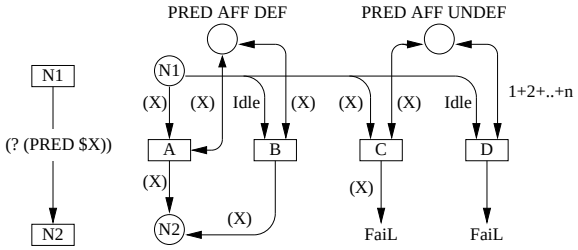

Figura 3: Teste de um predicado padrão.

Algoritmo

1. **Gera-se a permutação dos argumentos da expressão contida no arco. Por exemplo, se temos o seguinte teste afirmativo de predicado `(? (PRED $X $Y))` é gerada uma lista permutando-se cada variável do predicado entre definida `(X e Y)` e indefinida `(IDLE)`. No caso, a lista formada seria: `L = {(X, Y) (X, IDLE) (IDLE, Y) (IDLE, IDLE)}`.**
2. **Calcula-se o número de transição utilizadas na rede de Petri, sendo este número baseado no comprimento da lista de permutações. Este número é definido como 2 vezes o comprimento da lista de permutações, que para o caso do item anterior nos daria: comprimento de `L = 4`. Portanto, número de transições : `2 * 4 = 8`. Assim, a rede de Petri formada teria 8 transições das quais metade seria para compor o teste do predicado definido e a outra metade para testar o mesmo predicado agora indefinido.**
3. **São gerados os lugares que representam o predicado na forma padrão: `PRED_AFF_DEF, PRED_AFF_UNDEF`. Tais lugares serão fundidos a todos os outros lugares que aparecerem com o mesmo teste aplicado, do mesmo predicado, em outras sub-redes ou super-redes.**
4. **É somado aos lugares os nós representando os nós de entrada e de saída da sub-rede, fundidos na forma de uma rede hierárquica com a super-rede da KA.**
5. **Para i de 1 até o número_de_transições / 2 faça:**

   5.1. Cria-se um arco do nó de entrada da sub-rede `(In)` à transição i, cuja expressão é o elemento i da lista de permutações. Cria-se outro arco com a mesma expressão, ainda partindo-se do lugar `(In)` para a transição de número `((número_de_transições / 2) + 1)`.

   5.2. Para o lugar `(Out)` da sub-rede, cria-se um arco da transição `i` para este lugar com a expressão referente ao primeiro elemento da permutação.

   5.3. Para o lugar `AFF_DEF`, cria-se um arco de transição `i` para o lugar `AFF_DEF` cuja expressão é o primeiro elemento da permutação.

   5.4. Para o lugar `AFF_UNDEF`, são criadas relações a partir da segunda metade do número de transições, da transição `(i + (número_de_transições / 2))` para o lugar `AFF_UNDEF` com a expressão `i` da lista de permutações.

Codificação em Lisp (Operator - ?)

```
(defun regra-teste-aff (no1 no2 exp rpc))
(setf permutacao (permut (rest exp)))
(setf ntrans (* 2 (length permutacao)))
(setf pred (first exp))
(setf affdef (string-concatena pred "_AFF_DEF"))
(setf affundef (string-concatena pred "_AFF_UNDEF"))
(setf lugares (list no1 no2 affdef affundef "FAIL"))
(setf trans nil)
(do ( (i 1 (1+ i)) )
( (> i ntrans) )
(setf trans (append trans (list i))))
(rpetric lugares trans rpc)
; para o lugar no1
(do ( (j 1 (1+ j)) )
( (> j (/ ntrans 2)) )
(progn
(rpetric-atribui-arco-lt rpc no1 j (lista-elemento permutacao
(1- j)))
(rpetric-arco-lt rpc no1 j)
(rpetric-atribui-arco-lt rpc no1 (+ j (/ ntrans 2) )
(lista-elemento permutacao (1- j)))
)
)
; para o lugar no2
(do ( (k 1 (1+ k)) )
( (> k (/ ntrans 2)) )
(rpetric-atribui-arco-tl rpc k no2 (lista-elemento permutacao
0))
)
; para o lugar AFF_DEF
(do ( (k 1 (1+ k)) )
( (> k (/ ntrans 2)) )
(rpetric-atribui-arco-tl rpc k affdef (lista-elemento
permutacao 0))
(rpetric-atribui-arco-lt rpc affdef k (lista-elemento
permutacao 0))
)
; para o lugar AFF_UNDEF
(do ( (k (1+ (/ ntrans 2)) (1+ k)) )
( (> k ntrans) )
(rpetric-atribui-arco-tl rpc k affundef
(lista-elemento permutacao (- k (1+ (/ ntrans 2)))))
(rpetric-atribui-arco-lt rpc affundef k
(lista-elemento permutacao (- k (1+ (/ ntrans 2)))))
)
; para o lugar FAIL
(do ( (k (1+ (/ ntrans 2)) (1+ k)) )
( (> k ntrans) )
(rpetric-atribui-arco-tl rpc k "FAIL"
(lista-elemento permutacao (- k (1+ (/ ntrans 2)))))
)
)
```

5 ESTRUTURA GERAL DO CONVERSOR

O modelo de desenvolvimento do conversor seguiu a rotina convencional do modelo em cascata, onde cada fase deveria ser satisfeita para dar início a fase seguinte.

A divisão lógica do conversor é a seguinte:

1. **Entrada: é responsável por ler o arquivo texto que expressa o programa escrito em PRS (formato .opf) e transformá-lo em uma lista manipulável pelo núcleo do conversor.**

2. **Tabelas: são as estruturas utilitárias que servem de base para o núcleo (kernel) do sistema compor as redes e definir todos os parâmetros durante a fase de análise, a fim de que todas as informações necessárias à produção da saída estejam disponíveis na fase de síntese.**

3. **Núcleo (kernel): é o conversor propriamente dito, onde estão todas as funções responsáveis por conduzir a lógica em um plano mais geral.**

4. **Saída (xml): comporta todas as funções e variáveis utilizadas para gerar a saída do conversor (a rede de Petri obtida) de acordo com a** *Document Type Definition* **(DTD) do** *CPN-Tools***.**

Uma visão gráfica do conversor é mostrada na Figura 4.

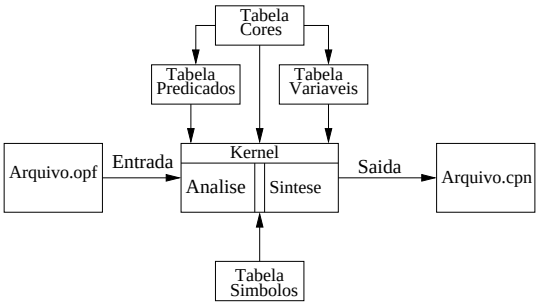

Figura 4: Arquitetura Geral do Conversor.

5.1 Fase de Análise do Conversor

A fase de análise do conversor é aquela que se preocupa principalmente com o tratamento do *arquivo.opf*, no formato de texto, no que diz respeito à sua estrutura. Faz a conversão deste arquivo para o formato de lista, a fim de ser melhor manipulado pelo lisp.

Trata também dos aspectos concernentes às tabelas de predicados, variáveis, símbolos e cores. E por fim, tem seu término obtido quando consegue (se o arquivo.opf assim o permitir) extrair do arquivo `.opf`, agora em formato de lista, os requisitos necessários para preencher todas as tabelas: predicados, variáveis, números de argumentos com seus domínios, etc. Estes dados são fundamentais para a geração das redes de Petri.

A fase de análise faz basicamente o seguinte:

1. **Lê o o programa de entrada PRS, em formato de texto. e,**

2. **Trata as tabelas (criação e manipulação).**

No que diz respeito ao item 1, o conversor transforma o arquivo.opf em um conjunto de listas, e em seguida faz uma limpeza delas a fim de retirar o que for irrelevante.

Varrendo o arquivo.opf, o conversor cria uma lista de KAS chamada `lkas` que armazenará as listas representantes das KAS; para cada KA tem-se uma lista, que é sublista de `lkas`. Cria também uma lista de variáveis, chamada `lvar`, associada a cada KA. Ou seja, nesse momente há a criação das listas de KAs e suas respectivas listas de variávies.

Com relação ao item 2, o conversor agora irá proceder à criação das tabelas. Primeiramente, ele cria a tabela de `Símbolos`, que é a representação das palavras reservadas da linguagem PRS, por exemplo: nomes de campos chaves como `INVOCATION, CONTEXT`; operadores como `“? “,”!”`. Enfim, tudo o que disser respeito à sintaxe do PRS.

Em seguida procede à criação da tabela de Predicados Gerais e a tabela de Variáveis, por KA. O conversor necessita de uma única tabela de predicados com seus respectivos nomes, número de argumentos e domínios dos argumentos - um predicado que apareça em uma KA, poderá aparecer em uma outra KA e será o mesmo em todo o sistema PRS. Com relação às variáveis, há a necessidade de se conhecerem as variáveis por cada KA, uma vez que os predicados terão em seus argumentos as variáveis daquela KA em particular. Isso é necessário para a montagem da rede de Petri e a correta aplicação das regras de conversão, bem como a propagação dessas variávies ao longo da rede.

O arquivo.opf é percorrido para extrair do mesmo todos os predicados presentes neste, com seus respectivos números de argumentos. Por exemplo, se o método responsável por percorrer o arquivo encontrar no campo `INVOCATION` da KA uma declaração do tipo `(INVOCATION(!(IR-PARA @R @P)))`, poderá extrair algumas informação relevantes:

1. **Como já foi criada a tabela de símbolos com as palavras reservadas do PRS, ele já sabe de antemão que a palavra `INVOCATION` é reservada, o mesmo valendo para `“!”`.**

2. **Após um operador `“!”` seguido de um abre parêntese, a função sabe que o nome que virá posteriormente é um predicado; no caso, `IR-PARA` é um predicado e automaticamente passará a figurar na tabela de predicados gerais;**

3. **Seguindo o arquivo, após ter determinado o nome do predicado, em se achando um caracter `@` ou `$`, o conversor verá que se trata de uma variável, guardando-a automaticamente na tabela de variáveis. A tabela de variáveis é armazenada na lista de variávies, a `lvar`. Ou seja, cada argumento desta lista (`lvar`) terá uma tabela de variáveis de uma KA. Assim procede até extrair todos os dados referentes a predicados e variáveis.**

Em seguida, o conversor procede à criação da tabela de cores que na verdade são os tipos dos argumentos dos predicados e variáveis. Por exemplo, se foi determinado que o predicado `(IR-PARA @R @P)` é assim composto: `(IR-PARA (A,B,C)(1,2,3))`, significa que a variavel `@R` terá como domínio `A,B` e `C`, que pertencerão a uma cor `P`, por exemplo; `@P` terá como domínio `1,2` e `3`, de cor `I`; então o predicado `IR-PARA` terá uma cor `T`, por exemplo. A

```
      Tabela de Predicado:
NOME      DOMÍNIO        No.ARG. COR
IRPARA   ((A B C) (1 2 3)) (2)   (T)
BOMBA    ((ON OFF))        (1)   (R)
CAIXA    ((B M C))         (1)   (W)
MONITORA (())              ( )   (P)
```

Figura 5: Tabela de Predicados

```
      Tabela de Variáveis:
NOME DOMÍNIO  COR PREDICADO
@R  ((A B C)) (P) (IRPARA 1)*
@P  ((1 2 3)) (I) (IRPARA 2)
$X  ((O F))   (D) (MOVER 1,ATENDER 2)
*Significa que a variável @R aparece
 no predicado IRPARA no 1o. argumento.
```

Figura 6: Tabela de Variáveis

determinação de todas as cores dos predicados e variáveis é chamada de geração de domínio.

Assim, o arquivo.opf é percorrido tantas vezes quantas forem necessárias até que estejam determinados todos os predicados e todas as variáveis, com as informações necessárias. O critério de parada dessa varredura é a não mudança nos entes envolvidos.

Portanto, ao final da fase de análise, o conversor deverá ser capaz de gerar uma saída do tipo: Figuras 5 e 6; lembrando que a tabela de predicados é única para todas as KAs enquanto que as tabelas de variáveis não o são: é por KA.

Assim, se o conversor puder preencher as tabelas com todas os requisitos necessários, a fase de análise está terminada e a próxima fase, a de síntese, terá início.

5.2 Fase de Síntese do Conversor

A fase de Síntese, sucintamente, é responsável pela geração da Rede de Petri Colorida em um formato interno. Este formato interno será convertido em um outro formato (a saída) para que possa ser lido pela ferramenta de CPN que utilzamos neste trabalho.

Para iniciar seus trabalhos, a fase de síntese necessita das informações da fase anterior, para lhe servirem de entrada, Figura 7.

As redes de Petri são armazenada na estrutura de dados "Rpetric" e representadas hierarquicamente em três níveis de abstração: a Meta-Rede, as Super-Redes e as Sub-Redes.

```
Lista de Ka's: (ka1, ka2,....., kai)
Lista de tabela de Variáveis:
 ((var1, var2,....,vari)
  (var1, var2,....,vari)
  (var1, var2,....,vari))
Tabela geral de Predicados:
  (predA, predB,..., predX)
Tabela de Símbolos:
  (''='',''!'', ''and'', ...)
```

Figura 7: Resultado da Fase de Análise

A Meta-Rede é uma tabela onde estão armazenadas todas as informações referentes às redes geradas no sistemas. É o nível mais alto possuindo visibilidade para as outras redes, conforme Figura 8

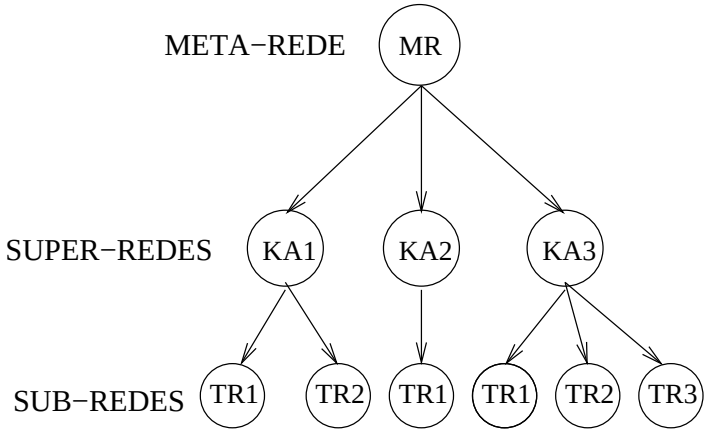

Figura 8: Hierarquização das Redes de Petri.

Para cada KA encontrada na lista de KAs (`lkas`) é criada uma Super-Rede que descreve o modelo geral da KA, onde cada transição dessa rede dá origem a uma Sub-Rede. Essa foi a forma utilizada para se fazer a construção hierárquica das redes de Petri.

Assim sendo, uma Meta-Rede tem tantas Super-Redes quantas forem as KAS presentes no sistema. Por sua vez, uma Super-Rede possuirá tantas Sub-Redes quantas forem as transições presentes na Super-Rede. Como uma transição na Super-Rede corresponde a um arco na KA, as Sub-Redes serão constuídas de acordo com a regra de conversão referente àquele tipo de arco.

Como vimos, o conversor cria primeiramente a Meta-Rede. A Meta-Rede armazena os dados que descrevem o ambiente global do sistema, como os predicados gerais, os lugares comuns compartilhados por todas as Super-Redes e Sub-Redes, os tipos de cores, expressões que geraram as Sub-Redes, etc.

Para cada KA encontrada na lista de KAs extraída do arquivo .opf, é criada uma Super-Rede que descreve o modelo geral da KA com suas transições representando suas redes hierárquicas (Sub-Redes) que possivelmente serão construídas a partir da aplicação de alguma regra sobre os argumentos inscritos como parâmetros da regra.

Para cada KA encontrada na lista de KAs (ka1, ka2,..., kai) é criada uma rede "`Rpetric`" sendo seus lugares os nós encontrados no elemento ka(x) através da função que manipula a estrutura `Rpetric`. Enquanto que as transições são os arcos do arquivo .opf. Com isso é gerado dinamicamente um identificador `Id` para essa Super-Rede.

Com relação à criação das Sub-Redes, procedemos à aplicação das regras de Conversão. Para cada Super-Rede é percorrida a lista de transições para identificá-las, caso existam, e assim definir qual a regra de conversão a ser aplicada. Daí, aplica-se a regra gerando uma Sub-Rede e uma nova entrada na tabela Meta-Rede (alimentando os

dados da Meta-Rede).

Em seguida, passamos à etapa de configuração da Rede de Petri, começando com a identificação das cores da mesma. Baseado nos domínios das variáveis e dos argumentos dos predicados são definidas as cores e suas relações (composição, pertinência, etc), gerando-se parâmetros para as variáveis que as definem junto à rede de Petri.

Em seguida, define-se o estado inicial da rede de Petri baseado no predicado geral e na forma como estes são representados e que valores (marcas) iniciais deverão possuir.

Com isso, o conversor definiu a rede de Petri internamente com todos os parâmetros necessários. Ou seja, nesse momento a tabela Meta-Rede já é a rede representativa do sistema.

A etapa seguinte diz respeito à geração da saída externa da rede. Será transformada essa representação interna da rede de Petri em um formato que seja possível ler em uma ferramenta para analisar rede de Petri coloridas, no caso o `CPN TOOLS`, baseado na `DTD` (Definição de Tipo de Documento) do mesmo.

6 ESTRUTURAS DE DADO USADAS E EXEMPLO PRÁTICO

A linguagem de programação utilizada para a implementação do conversor foi o *COMMON LISP*, dialeto padrão do LISP. Esta escolha deveu-se principalmente pelo fato do LISP ter um comportamento orientado ao tratamento de listas, o que é basicamente semelhante ao formato dos arquivos .opf (PRS). Porém, muitos dos recursos necessários para se implementar o conversor, principalmente as estruturas de dados, não são nativos à linguagem. Assim sendo, várias funções precisaram ser desenvolvidas, sendo as principais:

1. **LISTA - Apesar do LISP ser orientado basicamente ao uso de listas como sua estrutura primitiva, foi criada uma função LISTA própria para o conversor.**

2. **TABELA - Há na linguagem LISP uma estrutura chamada *hash-table*. Esta oferece uma interface que define o acesso para a leitura e escrita de posições da tabela *hash* com base na chave e na função interna de *hash*. Com isso, definiu-se para o conversor uma espécie de tabela `hash` aninhada, em que a função *hash* aplicada a uma entrada leva, na verdade, a outra *hash*.**

3. **RPETRIC - A estrutura Tabela guarda informações a respeito da relação dos elementos do conjunto de linhas com elementos do conjunto de colunas. Porém, em alguns momentos há a necessidade, em casos mais complexos, em que a informação é definida exclusivamente para uma dada linha ou coluna, ou até mesmo, para a própria tabela. Como uma rede de Petri necessita de mais informações a armazenar, como dados sobre transições, cores, etc, criou-se a estrutura de dados *Rpetric* a fim de cumprir essa função.**

6.1 Exemplo

Apresentaremos a seguir, um exemplo ilustrativo a fim de mostrar alguns resultados preliminares da abordagem acima referida. Trata-se de um sistema de caixa d'água que é composto de uma caixa d'água abastecida por uma cisterna. A Figura 9 mostra a KA do sistema. O objetivo é monitorar continuamente o sistema para que não falte água, no nível desejado, na caixa d'água.

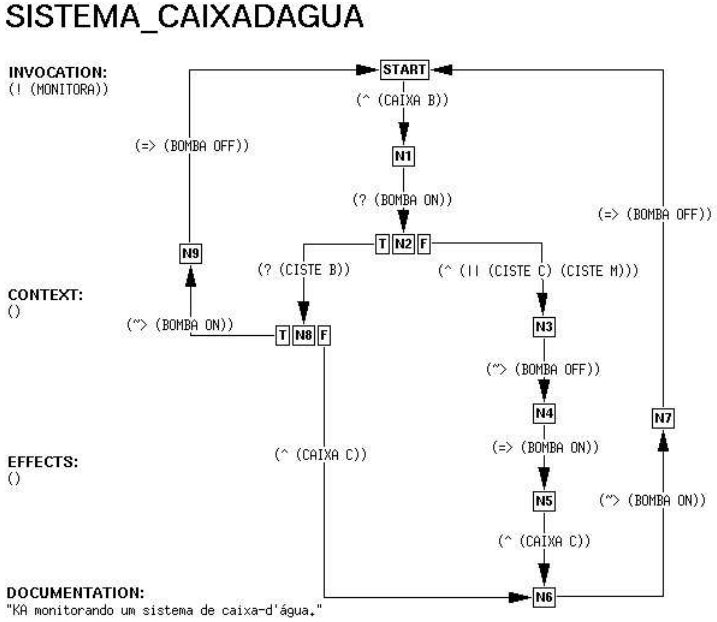

Figura 9: Sistema Caixa d'água.

Como ilustrado na figura, três predicados fazem parte da KA:

1. **`CAIXA`, representando a caixa d'água, com os argumentos `B` e `C` informando se a mesma está com seu nível baixo ou está cheia;**

2. **`BOMBA`, representando a bomba d'água, com os argumentos `ON` e `OFF`, ligada ou desligada;**

3. **`CISTE`, representando a cisterna do sistema, com os argumentos `B`, `M` e `C` informando se a cisterna está com seu nível de água baixo, médio ou está cheia.**

O sistema fica esperando que apareça no banco de dados do PRS o predicado informando que a caixa d'água está com o nível baixo (`(^(CAIXA B))`). Uma vez surgida essa informação, é testado se a bomba d'água está ligada `(?(BOMBA ON))`. Se não, o sistema espera que a cisterna esteja com seu nível cheio ou médio, predicado `(^(||(CISTE C)(CISTE M)))`, para retirar a informação que a bomba está desligada (`(~>(BOMBA OFF))`), e informar que a bomba foi ligada, predicado `(=>(BOMBA ON ))`. A partir daí, o PRS espera até que algum sensor indique que a caixa d'água está cheia, predicado (`(^(CAIXA C))`, retirando, em seguida, a informação de bomba ligada (`(~>(BOMBA ON))`) e concluindo, para o sistema, que a bomba foi desligada, através da crença (`(~>(BOMBA OFF))`). Nesse ponto, o

sistema monitora volta a seu estado inicial, nó **START**, começando todo o processo.

Note que explanamos apenas o procedimento para quando o teste do nó **if-then-else**, nó **N2**, for falso **(F)**. O procedimento é análogo, quando o teste deste mesmo nó for verdadeiro **((T)**.

A rede de Petri colorida obtida automaticamente através do conversor é mostrada na Figura 10. Nela é mostrada a Super-Rede do sistema (no caso uma, devido o sistema possuir apenas uma KA). As Sub-redes são redes hierárquicas obtidas da Super-rede, em que cada transição conduz a uma delas.

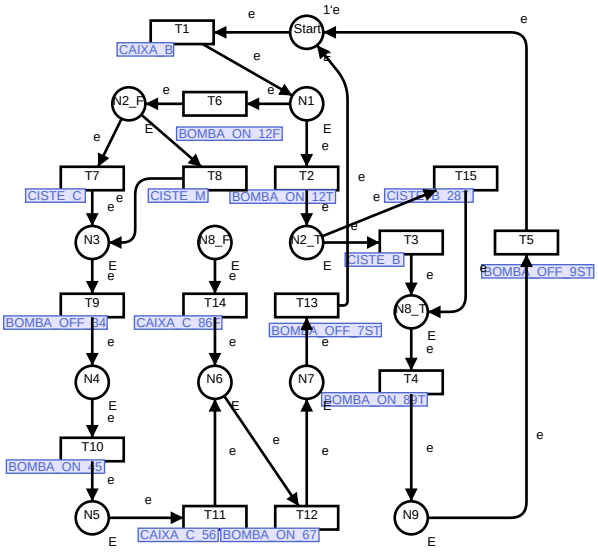

Figura 10: Sistema de caixa d'água

O programa PRS foi escrito com **OPRS**, *software* desenvolvido no grupo de Robótica e Inteligência Artificial do LAAS/CNRS (Toulouse, France). Para tratar as Redes de Petri Coloridas nós usamos o software **CPN Tools**, desenvolvido pelo Grupo CPN da Universidade de Aarhus, Dinamarca (Jensen, 2001). É uma ferramenta para edição, simulação e análise de CPNs. Para a análise, a ferramenta gera um grafo de acessibilidade (também conhecido como espaço de estados ou grafo de ocorrência). Este grafo permite, por inspeção dos estados, determinar todas as propriedades da rede (vivacidade, reinicabilidade,etc), apenas para uma marcação inicial bem precisa. O **CPN Tools** gera um relatório com informações de todas essas propriedades.

7 CONCLUSÃO

Como conclusões obtidas através da verificação formal feita através da análise da rede de Petri, podemos dizer que o sistema, conforme especificado, nunca entra em bloqueio (deadlock) porque sempre vai existir uma transição viva, ou seja, pode ser sensibilizada a partir dessa marcação inicial. Outra conclusão, é que a rede é reiniciável, podendo-se encontrar uma seqüencia de marcação que leva a rede de volta à marcação inicial.

Salientamos, ainda, que a grande vantagem dessa abordagem desenvolvida reside no fato de que a verificação formal não utiliza nenhuma linguagem de especificação particular. A verificão é feita através do próprio programa PRS, que é convertido automaticamente para as redes de Petri, evitando assim, possíveis erros caso a conversão fosse feita de forma manual. A grande desvantagem é exatamente a possibilidade de explosão da rede de Petri obtida, caso esta seja muito grande.

Assim, como mostrado, podemos usar os métodos e ferramentas de análise formal das redes de Petri para investigar as principais propriedades da rede gerada e, conseqüentemente, verificar indiretamente o programa PRS equivalente.

REFERÊNCIAS

Araújo, R. W. e Medeiros, A. A. D. (2004). *Verification of Procedural Reasoning System (PRS) Programs Using Colored Petri Nets (CPN)*, Kluwer Academic Publishers,USA. In Artificial Intelligence Applications and Innovations (editors: Bramer, Max and Devedzic, Vladan), pp. 421-433. ISBN 1-4020-8150-2.

Bench-Capon, T., Coenen, F., Nwana, H. e Paton, R. (Jun,1993). Two aspects of the validation and verification of knowledge-based system., *IEEE Expert*, pp. 76–81.

Georgeff, M. P. e Lansky, A. L. (1986). Procedural knowledge, *Proceedings of the IEEE 74*, Vol. 10, pp. 183–1398.

Ingrand, F. F. e Despouys, O. (2001). Extending procedural reasoning toward robot actions plannig, *IEEE ICRA 2001*, Seoul, South Korea.

Ingrand, F. F., Georgeff, M. P. e Rao, A. S. (December 1992). An architecture for real-time reasoning and sytem control, *IEEE Expert 7*, Vol. 6, pp. 34–44.

International, S. (1994). Shuttle malfunction handling. AI Center, 333,Ravenswood Avenue, Menlo Park,CA 94025, url: http://www.ai.sri.com/.

Jensen, K. (1994). *Coloured Petri Nets - Analysis Methods*, Vol. 2, Springer Verlag, Aarhus, Denamark.

Jensen, K. (1998). A brief introduction to coloured petri nets, *in* E. Brinksma (ed.), *Lecture Notes in Computer Science: Tools and Algorithms for the Construction and Analysis of Systems. Proceedings of the TACAS'97 Workshop, Enschede, The Netherlands 1997*, Vol. 1217, Springer-Verlag, pp. 201–208.

Jensen, K. (2001). Cpn tools - computer science department, university of aarhus.
*http://www.daimi.au.dk/ cpntools