Modelagem de programas PRS por redes de Petri coloridas

Adelardo Adelino Dantas de Medeiros
adelardo@leca.ufrn.br
Universidade Federal do Rio Grande do Norte – UFRN
Laboratório de Engenharia de Computação e Automação – LECA

Abstract – This paper presents some equivalence rules between PRS (a tool based on procedural reasoning) programs and colored Petri nets. This equivalence allows using the existing analysis methods for Petri nets to formally verify PRS programs.

Keywords: PRS; Procedural reasoning; Petri nets; Modeling.

1 Introdução

Sistemas automatizados para supervisão e diagnóstico são cada vez mais importantes no controle de processos industriais. Freqüentemente, estas tarefas são melhor tratadas não pelas técnicas de controle convencionais, mas sim por regras ou procedimentos que incorporam conhecimentos específicos da situação.

Um dos problemas mais difíceis no projeto de tais sistemas é o controle da sua execução em tempo real. Muitas das técnicas usuais de representação do conhecimento e inferência em Inteligência Artificial não são aplicáveis nesses sistemas, por não garantirem um tempo máximo de resposta.

Uma alternativa que vem sendo considerada nessas situações é o *raciocínio procedural* (Georgeff and Lansky 1986, Ingrand *et al*. 1992). O raciocínio procedural difere de outras representações usuais do conhecimento (regras, *frames*, etc.) por preservar informações de controle (ou seja, a seqüência de ações e testes) dentro de procedimentos ou planos, ao mesmo tempo que mantém alguns aspectos declarativos.

Uma das implementações mais difundidas dos conceitos do raciocínio procedural é o sistema PRS (*Procedural Reasoning System*). PRS tem sido adotado em diversas aplicações, tais como controle e supervisão de robôs móveis (Ingrand *et al*. 1996), na detecção de falhas em ônibus espaciais (SRI International 1994) e na gerência de redes de comunicação.

O preço a pagar pela versatilidade do raciocínio procedural aparece no fato de PRS não oferecer um mecanismo de verificação formal. Para suprir esta lacuna, uma possibilidade promissora é representar o conjunto de rotinas PRS por uma rede de Petri (RP) equivalente e utilizar os métodos existentes para verificação de propriedades estruturais e temporais das redes de Petri para validar o sistema PRS equivalente.

Neste artigo, nós apresentamos alguns resultados preliminares desta abordagem. O sistema PRS é descrito na seção 2, enquanto na seção 3 se faz uma rápida revisão dos conceitos fundamentais das redes de Petri coloridas. Na seção 4 se apresenta o principal da abordagem proposta, ou seja, as regras de equivalência entre rotinas PRS e RPs coloridas. Na seção 5 se revisam os principais métodos de verificação das RPs e na seção 6 são mostradas as perspectivas desta linha de pesquisa e os trabalhos futuros necessários para torná-la operacional.

2 A linguagem PRS

Certas características de PRS serão melhor explicadas na seção 4, quando da definição de sua representação por redes de Petri. No momento, pode-se dizer que um agente PRS consiste em:

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

Base de dados: contém fatos que representam a visão atual do mundo visto pelo sistema.

Conjunto das metas correntes: em PRS, as metas descrevem os objetivos e informam sobre a maneira de atingi-los. A meta para obter uma certa condição *C* se escreve `(! C)`; `(? C)`, para testar uma condição; `(^ C)`, para esperar até que uma condição seja verdadeira; `(# C)`, para preservar passivamente *C*; e `(% C)`, para manter ativamente *C*. Dois outros operadores permitem estabelecer `(=> C)` e excluir `(~> C)` uma informação na base de dados.

Grafo de intenções: um conjunto dinâmico de rotinas, estruturado sob a forma de um grafo, onde o sistema mantém informação em tempo real sobre o estado das rotinas escolhidas para execução e de suas sub-metas postadas.

Cada KA tem um corpo e algumas condições de invocação. Para exemplificar, apresenta-se na figura 1 uma KA para calcular fatoriais. Esta KA é considerada para possível execução quando um pedido para atingir uma meta `FATORIAL` é postado (como mostra o campo `INVOCATION`). Ela pode ser selecionada se as condições do campo `CONTEXT` forem satisfeitas. As condições podem conter variáveis a unificar (`$N` no caso). Há dois tipos de variáveis em PRS: as variáveis lógicas `$`*var* têm o comportamento clássico das variáveis em programação lógica (uma vez unificadas, não mudam mais de valor), enquanto que as variáveis de programa `@`*var* podem ser reatribuídas a todo instante.

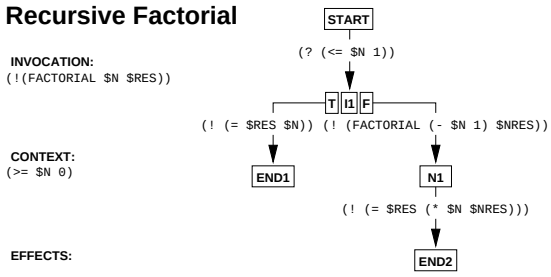

Figura 1: Um exemplo de rotina PRS (KA)

O corpo da KA descreve sua forma de execução. A execução inicia-se no nó START e prossegue seguindo os arcos através da

rede. Se mais de um arco partem de um nó, qualquer um deles pode ser atravessado. Para atravessar um arco, o sistema deve ou testar se a meta já foi estabelecida na base de dados ou lançar uma KA que atinge a meta associada ao arco: esta nova meta será incorporada ao grafo de intenções e deverá ser satisfeita para que a KA atual seja satisfeita. Se o sistema não consegue atingir nenhuma das metas associadas aos arcos que partem de um nó, a KA inteira falha. Mas quando um nó terminal (nó END) é atingido, a meta é considerada satisfeita e os eventuais fatos listados no campo EFECTS são concluídos na base de dados.

3 As redes de Petri coloridas

Não apresentaremos em detalhes a teoria das redes de Petri coloridas (RPCs): faremos apenas algumas observações intuitivas com base no exemplo da figura 2, que representa um caso simplificado de alocação de recursos (Jensen 1990). Uma formalização mais precisa pode ser encontrada na literatura (Murata 1989, Jensen 1994).

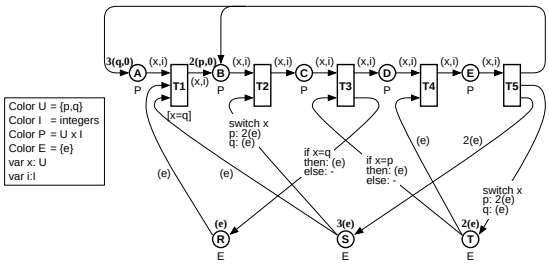

Figura 2: Um exemplo de rede de Petri colorida

Nas RPCs, cada lugar pode conter várias marcas, cada uma delas tendo um valor associado que denominamos a *cor* da marca. No exemplo, a cor das três marcas inicialmente presentes no lugar A é uma dupla (q,0). Cada lugar tem um tipo associado, que fixa as cores das marcas que ela pode conter. O lugar S, por exemplo, é do tipo E, o que implica que ele só pode conter marcas (e) (marcas sem cor). As variáveis também têm tipos associados.

Uma rede de Petri colorida pode ser considerada como uma versão estruturada de uma rede de Petri regular se o número de cores é finito. Como conseqüência, as propriedades que se pode provar com uma rede de Petri clássica também podem ser provadas com uma RPC de número de cores finito.

4 PRS e as redes de Petri

4.1 O princípio da modelagem

Para se determinar uma RPC equivalente a PRS, associa-se lugares da rede aos nós de PRS e transições da rede ou sub-redes aos arcos de PRS (fig. 3). Os deslocamentos da marca modelam a evolução da execução e sua cor contém o valor das variáveis atualmente definidas. Para que a RPC seja analisável formalmente, o número de cores deve ser finito, o que implica que só se pode ter variáveis "fechadas" (domínio de valores possíveis finito) no programa PRS que se quer modelar.

A cada arco são associados dois ramos na RPC: um correspondente ao caso onde a meta é atingida (a marca vai ao lugar que corresponde ao nó seguinte da KA) e o outro correspondente ao caso onde a meta fracassa (a destinação da marca varia).

Uma condição necessária para que uma sub-rede seja um modelo em RPC de um arco PRS é que ela possa ser reduzida a um encaminhador de marcas. A sub-rede consome a marca presente

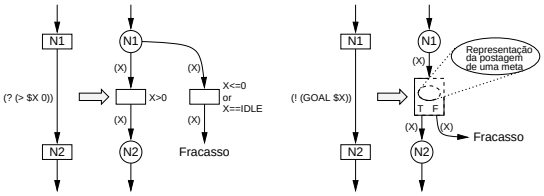

Figura 3: Dois exemplos de equivalência entre PRS e as RPCs

na sua entrada, mas garante a produção de uma marca em uma e uma só de suas duas saídas: aquela que modela a obtenção da meta ou aquela que modela seu fracasso (indicadas por T e F, respectivamente, nos nossos desenhos). De um modo geral, todo modelo de um arco deve poder ser reduzido aos elementos da figura 4, onde se garante que para toda marcação da rede há no máximo uma das duas transições (T ou F) validadas.

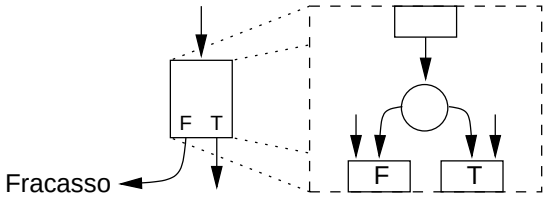

Figura 4: Característica essencial dos modelos dos arcos

4.2 A base de dados

Há diferentes tipos de predicados em PRS, cada um deles correspondendo a uma representação diferente. Por limitação de espaço, apresentaremos apenas a forma de representação dos predicados padrão. As outras representações são subconjuntos da representação dos predicados padrão.

Cada predicado padrão vai ser representado por quatro lugares, denominados AFF DEF, AFF UNDEF, NEG DEF e NEG UNDEF. Os lugares DEF et UNDEF correspondentes são complementares[1]. A presença de uma marca de uma certa cor em um lugar DEF indica que, para o valor correspondente, o predicado foi concluído (afirmativamente ou negativamente, caso se trate do lugar AFF ou do lugar NEG). De maneira similar, uma marca no lugar UNDEF indica que o predicado não foi concluído para esse valor. Para que a técnica dos lugares complementares seja aplicável, é necessário assegurar-se que:

- os lugares sejam limitados (tenham um número máximo de marcas), o que é verdade para um número finito de cores.
- para cada par de lugares complementares, a marcação inicial garanta a existência uma marca de cada cor em um dos dois lugares.
- na marcação inicial, não haja marcas de mesma cor simultaneamente nos lugares AFF et NEG.

Tomando-se como exemplo um predicado (EXIST *object colour*), onde *object* ∈ {BOOK, CAR}, *colour* ∈ {RED, BLUE, YELLOW}, indefinido no início para todas as combinações (*object colour*), a figura 5 mostra o estado da representação do predicado após as conclusões (=> (EXIST BOOK RED)), (=> (EXIST CAR GREEN)) e (=> (~(EXIST BOOK GREEN))).

[1] A marca que sai de um lugar complementar deve retornar ao mesmo lugar ou ir para o outro lugar complementar.

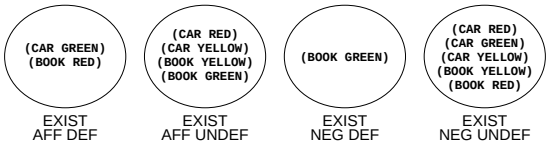

Figura 5: Modelagem de um predicado padrão

Os testes `(? (...))` sobre predicados padrão são representados pela RPC da figura 6, para um predicado genérico `(PRED` *var*`)`, $1 \leq var \leq n$. Mostramos apenas o modelo para o teste da afirmação `(? (PRED $X))`; o modelo do teste da negação `(? (~(PRED $X)))` é idêntico, com os lugares `PRED NEG DEF` e `PRED NEG UNDEF` substituindo os lugares `PRED AFF DEF` e `PRED AFF UNDEF`, respectivamente.

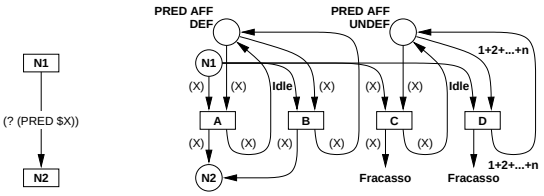

Figura 6: Teste de um predicado padrão

Toda marca retirada de um dos lugares que modelam `PRED` deve ser recolocada no mesmo lugar, pois um teste não altera o valor de um predicado. O comportamento do modelo de um arco de teste é diferente (transições diferentes são validadas) segundo o argumento do predicado. Por exemplo:

- O argumento já está unificado e para este valor o predicado está indefinido: o teste fracassa (transição `C`).
- O argumento não está unificado `(IDLE)` e há um valor para o qual o predicado está definido: o teste deu certo e transmite-se este valor (transição `B`). Este procedimento representa o mecanismo de unificação de variáveis.

Mostra-se na figura 7 a RPC equivalente à conclusão `(=> (~(PRED $X)))`. Para a conclusão da afirmação, ou seja, `(=> (PRED $X))`, basta trocar os lugares `AFF` e `NEG` correspondentes. Elimina-se uma eventual afirmação do predicado no momento de concluir sua negação (transição `C`). Uma conclusão com argumentos não unificados, o que constitui um erro, não muda nada no estado do predicado (transição `B`).

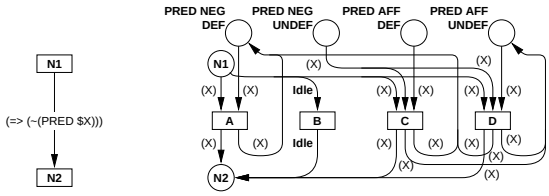

Figura 7: Conclusão de um predicado padrão na base de dados

O modelo da retirada da base de dados `(~> (...))` é mostrado na figura 8. Se a expressão a eliminar não está presente na base de dados (transição `B`), ou se o argumento não está unificado (transição `C`), o predicado não muda.

4.3 As variáveis

Todas as variáveis são transportadas nas marcas, de seu nascimento até o último momento onde elas serão utilizadas. A figura

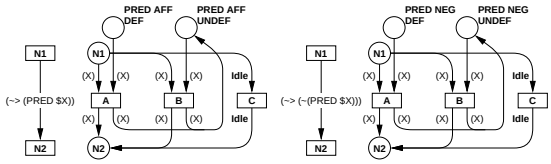

Figura 8: Retirada de um predicado padrão da base de dados

9 mostra um exemplo de transmissão de variáveis de uma KA pelas marcas de uma RPC. O modelos dos arcos foi simplificado porque o contexto de invocação da KA garante que as variáveis `$X` e `$Y` estão unificadas.

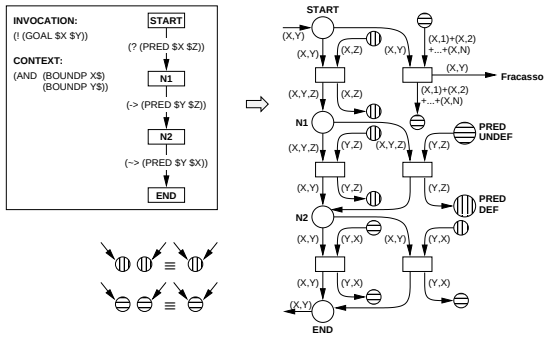

Figura 9: A representação das variáveis nas marcas

A marca será uma n-upla com tantos elementos quantas variáveis `PRS` a preservar. Os elementos da n-upla terão tantos valores possíveis quantos os valores possíveis para a variável `PRS` que eles representam, mais um `(IDLE)` para indicar que a variável ainda não foi unificada.

4.4 As metas

Quando uma meta é postada, ela pode ser satisfeita seja por consulta à base de dados, seja por chamada a uma KA que vai satisfazê-la. Para cada predicado que pode vir a ser postado como uma meta, deve-se ter, além dos lugares que representam o predicado na base de dados (ver seção 4.2), três outros lugares específicos das metas: `POST`, onde a rotina "cliente" assinala que a meta foi postada, e `SUCC` e `FAIL`, onde a rotina "servidora" indica o sucesso ou fracasso na obtenção da meta.

Na figura 10 apresenta-se um modelo mais completo da postagem de uma meta. No momento de postar a meta, a RPC coloca uma marca (com as cores dos argumentos da meta) no lugar `POST` e a execução desse ramo entra em um estado de espera (lugar `W`) de uma resposta quanto à obtenção da meta. Esta resposta se materializará pela aparição de uma marca no lugar `SUCC` ou no lugar `FAIL`.

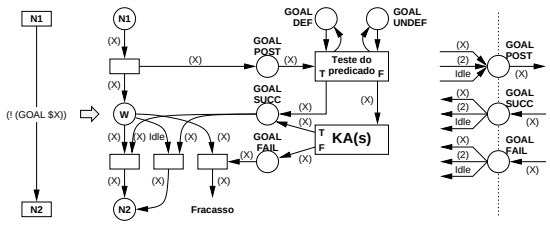

Figura 10: Postagem de uma meta

Após a postagem da meta, o modelo verifica se o predicado já não é satisfeito na base de dados, através de um teste como o da figura 6. Se a base de dados contém o predicado, a meta é

satisfeita; senão, são as eventuais KAs ativadas por essa meta que vão decidir sobre a satisfação ou não da meta (o modelo da chamada às KAs será visto posteriormente). Na figura 10 representou-se uma única rotina cliente, mas pode haver várias, como sugere o pequeno desenho à direita da figura.

4.5 Os nós

Normalmente, cada nó corresponde a um lugar na RPC, exceto para dois tipos especiais: os nós (`IF-THEN-ELSE`) e os nós que começam ou terminam uma execução em paralelo.

4.5.1 Os nós condição

A travessia de um arco que chega a um nó condição é sempre possível. Se a meta associada ao arco é atingida, a execução continua a partir da metade `T` do nó condição; senão, a partir da metade `F`. Este nó é representado por dois lugares, um para cada metade. Como cada modelo de arco deve gerar duas saídas possíveis para a marca, vamos dirigir o ramo que representa o sucesso na obtenção da meta para o lugar `T` e o ramo que modela o fracasso para o lugar `F`, como indicado na figura 11.

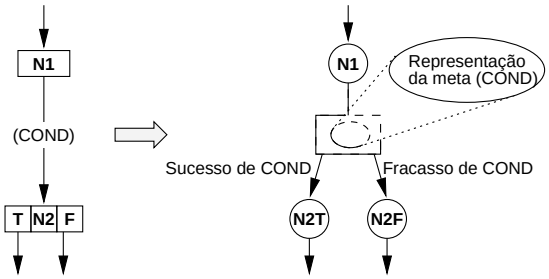

Figura 11: Modelo de um nó condição

4.5.2 O paralelismo

Um nó de separação (indicado por uma barra sob o retângulo) lança tantos ramos de execução em paralelo quantos forem os arcos partindo dele, enquanto que um nó de junção (indicado por uma barra sobre o retângulo) só continuará a execução da KA após a obtenção das metas associadas a todos os arcos que nele chegam. O fracasso de um ramo faz fracassar a KA inteira.

O modelo para esses nós, apresentado na figura 12, faz aparecer para cada nó, além do lugar normalmente associado ao nó, uma transição e tantos lugares suplementares quantos forem os ramos a serem executados em paralelo.

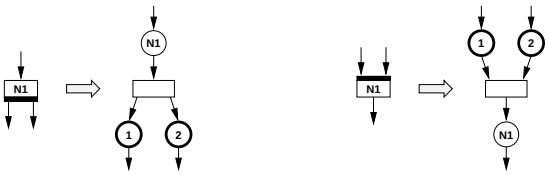

Figura 12: Modelos dos nós de início e fim de paralelismo

4.6 Os arcos

4.6.1 As solicitações de objetivos

As solicitações de objetivos (`! GOAL`) podem ser utilizadas com predicados definidos pelos usuários ou com primitivas.

Para os predicados definidos pelo usuário, o modelo da rotina de postagem de uma meta foi mostrado na figura 10. Falta explicar o mecanismo de escolha das KAs para atingir as metas que não são satisfeitas na base de dados, o que será feito na seção 4.9.

A maioria das primitivas utilizadas nas solicitações de objetivos correspondem a KAs cujas ações nunca falham (`PRINTF`, ...), e que portanto não têm maior interesse para a análise. Esses arcos podem ser substituídos por transições sempre satisfeitas. Uma exceção notável é a primitiva de atribuição (`=`): se a atribuição é sempre possível para as variáveis de programa, ela só o é para as variáveis lógicas ainda não unificadas (fig. 13).

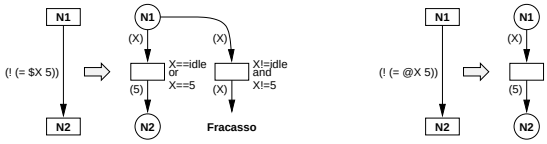

Figura 13: Modelo da atribuição de uma variável

4.6.2 As esperas

Um arco de espera nunca falha: no máximo, fica em espera eterna. Seu modelo prevê uma única transição, a ser disparada quando a condição for satisfeita (fig. 14).

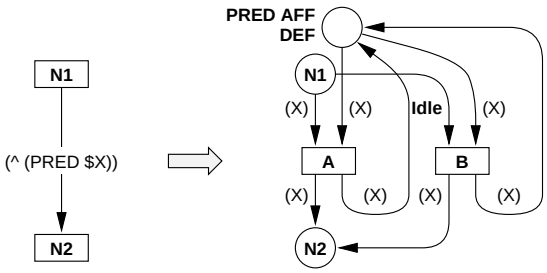

Figura 14: Modelo de um arco de espera

4.7 O não determinismo

Em PRS, quando vários arcos partem de um mesmo nó, o sistema escolhe aleatoriamente um dos arcos para tentar atravessar. Se não consegue, ele escolhe um outro arco e só considera que a KA fracassou após haver tentado todos os arcos. PRS não faz *backtracking*: se ele consegue atravessar um arco, mas fracassa na continuação deste ramo, ele não volta ao nó para testar os arcos não explorados.

Dado esse comportamento, é preciso conceber uma estrutura para garantir que o modelo teste todos os arcos antes de assinalar um erro. Essa estrutura é mostrada na figura 15 para um caso com dois arcos em conflito, mas é facilmente extensível para qualquer número de arcos.

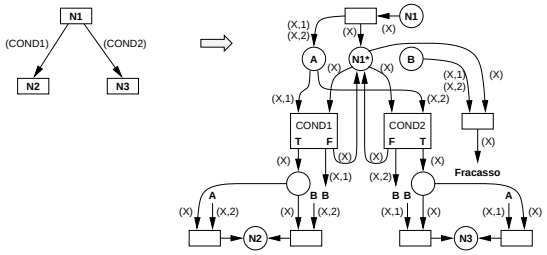

Figura 15: Nó com vários arcos que saem

Inicialmente, o lugar `A` contém 2 marcas numeradas, uma para cada arco. A única marca em `N1*` garante a execução de um

único arco por vez. Se a execução do arco que pegou a marca em N1* tiver sucesso, o modelo limpa os lugares A e B (consome todas as marcas dos outros ramos) e a execução continua. Em caso de fracasso, o modelo recoloca a marca em N1*, autorizando outro arco a ser testado, e põe a marca correspondente ao arco que falhou em B. Se todas as marcas se encontrarem em B, o modelo indica um erro.

4.8 As KAs

Toda RPC que modela uma KA tem um lugar de início, chamado INIT. Entre INIT e START há uma sub-rede que testa se o contexto de execução (o campo CONTEXT) da KA foi satisfeito. Isso equivale a acrescentar um arco suplementar na KA para fazer o papel do campo CONTEXT, o que não modifica a execução do programa (fig. 16). Da mesma forma, entre o lugar END, que faz o papel de todos os nós terminais da KA, e o lugar TERM, que modela o término com sucesso da execução da KA, há uma sub-rede que modela o campo EFFECTS.

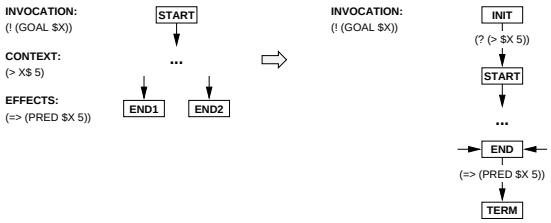

Figura 16: Equivalência entre certos campos da KA e arcos

Os modelos em RPC das KAs devem também conter um lugar FAIL, onde a aparição de uma marca indica que a KA fracassou. A saída em caso de erro (F) de todos os modelos dos arcos deve ser direcionada para esse lugar FAIL, exceto para as situações especiais dos arcos que conduzem a um nó IF-THEN-ELSE (seção 4.5) e dos arcos que partem de um mesmo nó (seção 4.7). A figura 17 mostra uma RPC equivalente a uma KA genérica. Os modelos dos arcos propriamente ditos não são detalhados e supõe-se que só há uma variável a transmitir (X) na KA.

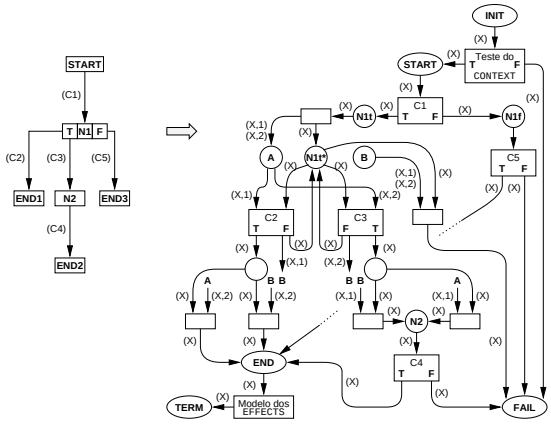

Figura 17: Exemplo de RPC equivalente a uma KA

4.9 A ativação das KAs

Para as KAs que reagem a fatos, uma simples transição detecta a aparição do fato na base de dados e põe uma marca em INIT. Para as KAs disparadas por metas, é preciso voltar ao modelo de postagem de uma meta (fig. 10). Há três casos distintos:

1. Nenhuma KA é disparada pela meta.
2. Só uma KA é disparada pela meta.
3. Várias KAs são disparadas pela meta.

O teste da obtenção de uma meta que não dispara KAs se resume ao teste da presença do predicado na base de dados (fig. 18).

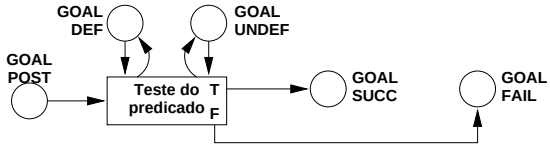

Figura 18: Meta com nenhuma KA disparável

Quando só há uma KA que satisfaz a meta, e se o predicado não foi estabelecido na base de dados, o sucesso ou fracasso da meta é indissociável do sucesso ou fracasso da KA (fig. 19).

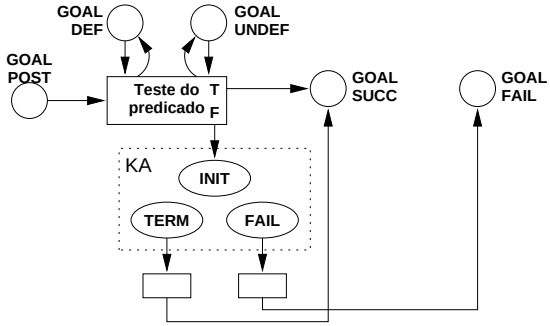

Figura 19: Meta com uma única KA disparável

Finalmente, quando há várias KAs disparáveis pela mesma meta, PRS vai fazer a escolha de maneira não determinística. Vai-se utilizar uma estrutura similar à da figura 15 para garantir que todas as KAs serão testadas antes de se assinalar um erro. A figura 20 mostra um exemplo com duas KAs.

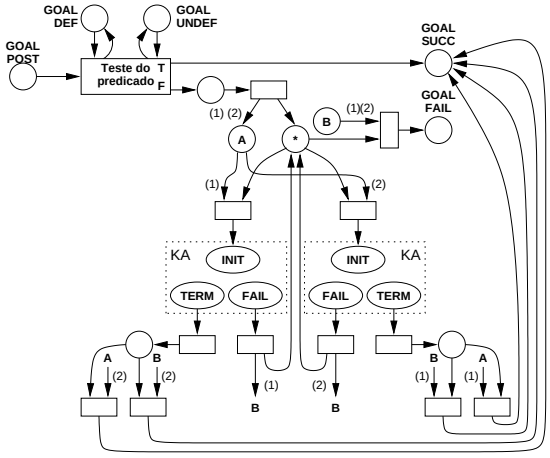

Figura 20: Meta com várias KAs disparáveis

4.10 As limitações

A maioria das limitações desse método de representação de PRS por RPCs está implícita nos diversos modelos apresentados. Listamos aqui algumas das mais notáveis:

Preservação de atributos: não se obteve equivalentes genéricos dos operadores de preservação passiva (# *C*) e ativa (% *C*) de um predicado, que felizmente não são muito utilizados em PRS.

Domínio fechado das variáveis: impede a utilização dos reais, dos inteiros sem limites e de todo outro tipo de dado não enumerável.

5 Métodos de verificação

Não vamos apresentar nesse artigo os métodos de verificação das redes de Petri coloridas, já consolidados na literatura principalmente nos trabalhos de Jensen (Jensen 1990, Jensen 1994). Sinteticamente, podemos dizer que as RPCs podem ser analisadas de três maneiras principais:

- por simulação;
- através do seu grafo de acessibilidade (também conhecido como espaço de estados ou grafo de ocorrência); e
- pelo método dos invariantes (de lugar ou de transição)

Os métodos de análise formal permitem provar um certo número de propriedades para as redes de Petri. Essas propriedades podem ser comportamentais (para uma certa marcação inicial) ou estruturais (que só dependem da estrutura da rede). Os invariantes estabelecem certas propriedades estruturais e o grafo de acessibilidade permite, por inspeção dos estados, determinar todas as propriedades da rede, mas apenas para uma marcação inicial bem precisa.

Para a RPC de nosso modelo, as propriedade que nos interessam são principalmente as provadas para uma marcação inicial definida, por duas razões:

- Há lugares complementares na rede, o que só dá sentido ao modelo caso se fixe uma marcação inicial coerente com a utilização desse artifício.
- Normalmente, o que nos interessa é poder ter garantias quanto ao comportamento do sistema face a situações (marcações) precisas.

As propriedades demonstradas para a RPC devem ser interpretadas sabendo que a rede modela um conjunto de KAs e a partir do que a marcação inicial representa. Um lugar não limitado em número de marcas, por exemplo, indica quase certamente um erro na escrita das KAs (como uma KA que se chama recursivamente sem limites). A existência de um bloqueio (quando não há nenhuma transição disparável e a rede "morre") pode indicar um erro de concepção ou, ao contrário, provar que o funcionamento do sistema está correto:

- Se o comportamento que se espera do sistema para uma dada marcação inicial é que ele entre em um ciclo sem fim, a existência de uma possibilidade de bloqueio indica um erro de projeto.
- Para uma marcação inicial onde há uma meta postada, é necessário que a RPC termine em bloqueio, com uma única marca em um dos lugares que indicam o sucesso ou fracasso da meta.

6 Conclusão e perspectivas

Foram apresentadas regras de conversão que permitem converter um subconjunto relativamente abrangente de PRS para redes de Petri coloridas, o que abre a possibilidade de utilização de técnicas e das ferramentas de verificação de redes de Petri, como o PROD (`http://saturn.tcs.hut.fi/pub/prod/`) na verificação de programas PRS.

É grande a tentação de, uma vez que se dispõe dessa equivalência, fazer a conversão manual dos programas PRS e verificar as propriedades da rede de Petri obtida. Mas a probabilidade de introdução de erros na conversão é grande, o que faz com que o esforço necessário para se garantir que a conversão manual não contém erros seja da mesma ordem de grandeza que a eliminação de eventuais erros diretamente no programa PRS original.

É uma constante na maioria dos trabalhos sobre verificação de sistemas ditos inteligentes (por exemplo, (Liu and Dillon 1991) e (Lee and O'Keefe 1994)) que a metodologia deve basear-se em ferramentas automatizadas. Portanto, o próximo passo na evolução desta linha de pesquisa é o desenvolvimento de um conversor automático de PRS para RPCs. Isso é possível porque todas as regras de equivalência apresentadas seguem leis de formação genéricas, aptas a serem incluídas em algoritmos.

Alguns trabalhos preliminares já foram feitos nesse sentido, de modo que se espera que dentro de algo tempo poderemos mostrar resultados e verificar a aplicabilidade da metodologia proposta em sistemas reais. O aspecto que mais nos preocupa, e que só poderá ser verificado quando se fizerem testes sobre um programa PRS complexo, é se a complexidade da RPC que modela o sistema não será tal que inviabilizará a utilização das ferramentas disponíveis para verificação de redes de Petri.

REFERÊNCIAS BIBLIOGRÁFICAS

Georgeff, M. P. and A. L. Lansky (1986). Procedural knowledge. *Proceedings of the IEEE* **74**(10), 1383–1398.

Ingrand, Fran cois Félix, Raja Chatila, Rachid Alami and Frédéric Robert (1996). PRS: A high level supervision and control language for autonomous mobile robots. In: *IEEE International Conference on Robotics and Automation*. Vol. 1. Minneapolis, Minnesota, USA. pp. 43–49.

Ingrand, François Félix, Michael P. Georgeff and Anand S. Rao (1992). An architecture for real-time reasoning and system control. *IEEE Expert* **7**(6), 34–44.

Jensen, Kurt (1990). Coloured petri nets: a high level language for system design and analysis. In: *Advances in Petri Nets* (G. Rozenberg, Ed.). Vol. 483 of *Lecture Notes in Computer Science*. pp. 342–416. Springer-Verlag.

Jensen, Kurt (1994). An introduction to the theoretical aspects of coloured petri nets. In: *A Decade of Concurrency* (J.W. de Bakker, W.-P. de Roever and G. Rozenberg, Eds.). Vol. 803 of *Lecture Notes in Computer Science*. pp. 230–272. Springer-Verlag.

Lee, Sunro and Robert M. O'Keefe (1994). Developing a strategy for expert system verification and validation. *IEEE Transactions on Systems, Man and Cybernetics* **24**(4), 643–655.

Liu, N. K. and T. Dillon (1991). An approach towards the verification of expert systems using numerical petri nets. *International Journal of Intelligent Systems* **6**, 255–276.

Murata, Tadao (1989). Petri nets: Properties, analyses and applications. *Proceedings of the IEEE* **77**(4), 541–580.

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