3
Arquitetura Proposta

O processo usual de verificação formal de algoritmos distribuídos e protocolos (figura 3.1) demanda normalmente um grande esforço por parte do projetista destes protocolos. Um motivos é a complexidade da especificação em sí, que pode ser causada pela complexidade inerente do protocolo, pela pouca afinidade do projetista com a linguagem do verificador ou pela incompatibilidade existente entre o que linguagem do verificador pode modelar e as características do protocolo a ser modelado. Além disso, as especificações das propriedades sobre um protocolo são normalmente fornecidas em lógica temporal e podem ser difíceis de serem especificadas pelos mesmos motivos da especificação do protocolo. Como resultado, podemos comprometer a interpretação dos contra-exemplos retornados pelo verificador que, de acordo com o modelo que está sendo verificado, podem ser extensos, conter ciclos, conter informação irrelevante ou até serem causados por erros de modelagem do protocolo.

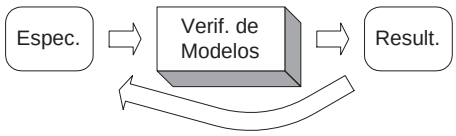

Figura 3.1: Modelo tradicional de verificação

Usualmente as linguagens de especificação dos verificadores são definidas num nível de abstração muito baixo, o que torna as especificações mais extensas e, por muitas vezes, ilegíveis. A implicação imediata disto é a desconfiança no que foi especificado, ou seja, se foi realmente especificado o que se desejava. Por transitividade, também passamos a questionar a veracidade do resultado obtido na verificação.

Motivados por estas dificuldades, propomos esta arquitetura que tem por objetivo simplificar o processo de verificação de formal de protocolos, em especial para a computação móvel, e algoritmos distribuídos. Devido à carência de linguagens com construções específicas para estes sistemas, propomos a linguagem LEP, de domínio específico, que contém construções de alto nível para facilitar a prototipação destes sistemas. Para validarmos estas

especificações, implementamos transformações que traduzem as construções de LEP para a linguagem de entrada do verificador formal escolhido. No caso dos verificadores de modelos, um contra-exemplo é retornado quando uma especificação não satisfaz uma determinada propriedade. Como este contra-exemplo não reflete as construções de LEP, mas sim o resultado das transformações, descrevemos de que maneira recuperamos estes contra-exemplos no nível de LEP para os verificadores de modelos utilizados. Seguindo a filosofia dos verificadores de modelos, todo o processo é feito de forma automática, sem o auxílio do usuário.

3.1 Descrição

A figura 3.2 descreve o processo de verificação de protocolos proposto. Inicialmente temos a especificação do protocolo em LEP, descrita na seção 3.2. Esta especificação contém a descrição da topologia de rede sobre a qual o protocolo é projetado, a especificação do protocolo propriamente dita e a especificação das propriedades a serem verificadas.

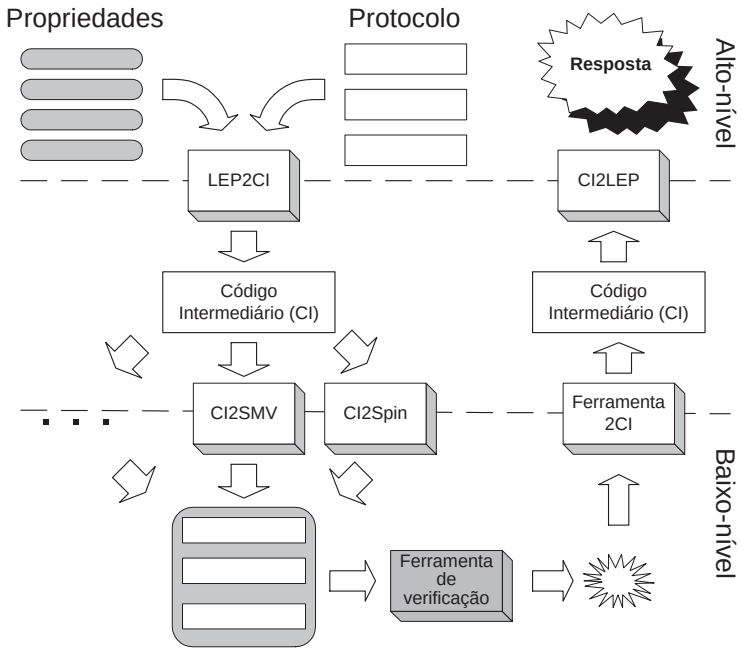

Figura 3.2: Descrição geral da arquitetura

A especificação fornecida pelo usuário é passada para o módulo LEP2CI, que a traduz para um código intermediário, o qual contém construções comuns à maioria das linguagens de especificação. O código intermediário obtido é passado para algum dos módulos de conversão de forma que a associada ferramenta de verificação possa ser utilizada com o modelo convertido. Deste ponto em diante temos o tradicional processo de verificação de modelos.

O resultado desta verificação é então passado a outro módulo, que fará a interpretação do resultado do verificador, gerando o código intermediário respectivo. Este, por sua vez, é processado para que a resposta ao usuário seja dada em termos de LEP. Todas as traduções entre as especificações são feitas através da linguagem de transformação de código TXL (Tree Transformation Language) (Cor04) sem interação com o usuário.

3.2 LEP

LEP (Linguagem para a Especificação de Protocolos) é a interface entre o usuário e o sistema proposto para a verificação de protocolos móveis e sistemas distribuídos. O objetivo não é criar uma linguagem totalmente nova, mas sim propor uma que use estruturas já conhecidas de outras linguagens de especificação como SDL (Z.102) Promela (Hol97) e MMC (YRS03), além de adicionar alguns facilitadores (pronomes) que simplificam o processo de especificação de protocolos e algoritmos distribuídos.

3.2.1 Descrição

LEP é uma linguagem de domínio específico para a especificação de protocolos e algoritmos distribuídos. É baseada em processos e combina os conceitos de comandos guardados de CCS (Mil89), sobrecarga de nomes de π-calculus (MPW92) e pronomes (adaptado da pesquisa em OO sobre referências a objetos proposta em (CLN01)). Pronomes podem ser vistos como forma geral e uniforme de referenciarmos um conjunto de elementos (nós numa rede), tornando a especificação mais compacta, legível e precisa.

Uma especificação em LEP pode ser dividida em 3 partes: a primeira trata da definição da topologia sobre a qual a especificação se baseará; a segunda refere-se a especificação do modelo dos protocolos, e; a última à linguagem de especificação de propriedades a serem verificadas.

Uma topologia pode ser fornecida de 3 maneiras: indicando explicitamente as conexões da rede; através de uma gramática de grafos com atributos, e; utilizando macros para gramáticas de topologias pré-definidas. Estas variações são detalhadas na seção 3.2.2.

A segunda parte da especificação, que é detalhada na seção 3.2.3, é baseada em comandos guardados de CCS (Mil89), sobrecarga de nomes (identificadores podem ser dados ou canais de mensagens) de pi-calculus (MPW92) e pronomes. Pronomes, descritos à seguir, são tratados aqui como uma maneira de se fazer referência de maneira geral, uniforme e simplificada

a um conjunto de elementos no protocolo. Por exemplo, uma comunicação comum entre 2 processos descrita na linguagem de um verificador (baseado em cálculo de processos) tem o seguinte formato:

P_1: c!x
P_2: c?y

, onde c é o canal de comunicação, x é a mensagem a ser enviada e y é um parâmetro que receberá a mensagem enviada. Se queremos enviar a mesma mensagem para todos os nós, uma maneira seria através de iteração no conjunto de elementos:

P_1: for i=1 to N do
c[i]!x

, onde i é o índice para o número de elementos N do sistema. Usando LEP, escrevemos simplesmente:

P_1: everyone!x

, onde *everyone*[1] representa todos os elementos no conjunto. Neste caso, o agente (elemento que envia a mensagem) não precisa estar diretamente conectado a todos os nós da rede. Quando este não está, a mensagem é encaminhada por "inundação"entre os vizinhos do agente de forma a enviar a mensagem a todos os nós alcançáveis por este.

Com o pronome *everyone* também podemos pensar em termos de recebimento:

P_2: everyone?y

, onde podemos imaginar que P_2 é um ponto de sincronismo de um processo mestre (barreira). Neste caso, um processo que execute P_2 esperará por uma mensagem y enviada por todos os nós da rede, e não apenas os alcançáveis. Isto porque num ambiente distribuído, de acordo com o comportamento dinâmico da rede, um nó não sabería precisar qual é o conjunto de nós alcançáveis por este num dado momento.

Destas duas (2) situações podemos perceber que não existe a noção de canal pura e simples em LEP. Cada referência a um ou vários nós na rede, por si só, já faz o papel de dado ou canal, dependendo de onde este ocorre no comando.

[1]De forma a facilitar a comparação de LEP com outras linguagens de especificação de sistemas concorrentes, todas as construções de LEP foram definidas em inglês.

A parte de LEP que lida com a descrição de propriedades sobre o modelo é baseada em lógicas temporais[2], como normalmente encontramos nos diversos verificadores de modelos existentes. Entretanto, também simplificaremos esta parte da especificação de 2 maneiras: permitindo a utilização de pronomes nas consultas e utilizando padrões de especificação de consulta encontrados em (DAC99). Na verdade, podemos mapear estes padrões de consulta para os pronomes. Por exemplo, o padrão *Absence* pode ser descrito da seguinte forma em LTL:

$$F_1\text{: } [] \text{ (not P)}$$

, onde [] é o operador modal que indica que sempre teremos a negação de *P*, ou seja, nunca teremos *P*.

Se queremos falar de uma mensagem de erro, podemos escrever:

$$F_1\text{: } [] \text{ (not c!error)}$$

, onde *c* é um canal e *error* é o tipo da mensagem enviada. Em LEP podemos escrever:

$$P_1\text{: } [] \text{ (none!error)}$$

, onde *none* é o pronome que indica que nenhum elemento se apresentará em tal cenário.

Detalhes da especificação de propriedades de um modelo em LEP são apresentados na seção 3.2.4.

Dada a necessidade de confiança na especificação (especificação × intenção, citado na seção 2.3), quanto mais preciso for a descrição, mais próximas estarão as palavras especificação e intenção. Ou seja, características inerentes de um protocolo, como tempo-real e incerteza (probabilidade), precisam se contempladas pela linguagem de especificação. LEP ainda não possui tais mecanismos, apesar de termos artifícios que simulam estes conceitos. Estes serão utilizados na descrição dos casos de uso (seção 4.1). Entretanto, para maior fidelidade, estes conceitos serão incorporados em trabalhos futuros.

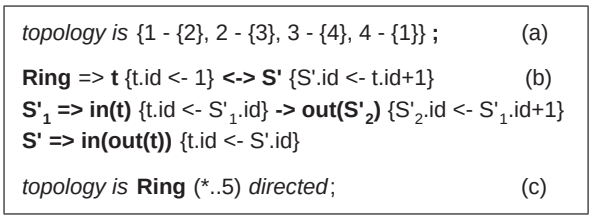

Figura 3.3: Especificações de topologias em LEP

3.2.2 Especificação das Topologias

A figura 3.3 apresenta três maneiras distintas de se especificar topologias similares em LEP.

A especificação 3.3.a define explicitamente as conexões entre nós na rede. Nó 1 tem vizinhança (nós conectados diretamente a ele) 2, nó 2 tem vizinhança 3, 3 tem 4 e 4 tem 1. Ou seja, especificamos um anel.

A especificação 3.3.b expressa uma topologia em anel rotulada pelo atributo herdado *id* através de uma gramática de grafos com atributos. Como esta é uma definição geral de anéis, sempre que definimos manualmente uma gramática precisamos instanciá-la, como na especificação 3.3.c. Sobre o rótulo *id*, na primeira regra de 3.3.b, este recebe o valor 1 e o elemento S' recebe este valor mais um (+ 1), e assim sucessivamente até que o último elemento seja adicionado.

Atribuições entre chaves ({}) referem-se a manipulação de atributos, enquanto que elementos fora das chaves como setas ($\leftarrow, \rightarrow, \leftrightarrow$) e identificadores referem-se à gramática de grafos. **Ring**, **S** e **S'** são não-terminais, **t** é um terminal rotulado, $\Rightarrow$ conecta um não-terminal com seu lado direito e **in** e **out** indicam os nós de entrada e saída, respectivamente, de uma regra numa gramática de grafos. A notação gráfica ilustrativa e um exemplo mais detalhado do uso destas gramáticas são dados à seguir (seção 3.2.2).

A especificação 3.3.c, quando utilizada sozinha, supõe a existência de uma macro para uma topologia pré-definida *Ring*. Esta especifica anéis com no máximo cinco (5) nós. Nesta especificação, a palavra reservada *directed* é um parâmetro da topologia e indica que as arestas são orientadas, como na especificação 3.3.a. *Undirected* seria o oposto, ou seja, para toda aresta vale a ida e a volta. Os parâmetros de uma topologia podem ser *(un)reliable*, *(un)directed*, *(un)secure*, *(dis)connected*, *dynamic* e *static*. Uma topologia é *reliable* quando suas arestas e nós não falham, ou seja, as mensagens não são perdidas na rede. Uma topologia é *secure* quando as mensagens trafegadas

[2]Nos baseamos em μ-calculus que engloba grande parte das lógicas temporais. Entretanto, na implementação existente, LTL foi adotada como a linguagem de especificação de propriedades

pela rede não se corrompem ou são corrompidas. Uma topologia é *disconnected* quando é permitido que não exista caminho entre dois nós numa rede. Quando uma topologia é *dynamic*, conexões entre nós podem ser inseridas ou removidas transparentemente na rede, seguindo algum modelo, de forma a simular movimentações. Numa topologia *static*, não temos a geração transparente das movimentações. *Reliable*, *undirected*, *secure*, *connected* e *static* são os valores padrões destes parâmetros.

Sobre os parâmetros *dynamic* e *static*, estes servem para indicar se a movimentação de nós será tratada de forma transparente ou não, respectivamente. Por exemplo, a modelagem da movimentação de agentes num protocolo para redes estruturadas normalmente é feita de forma explícita, ou seja, o agente sinaliza explicitamente a intenção de movimento (p. ex, mudança de célula de cobertura). Neste caso o parâmetro utilizado seria *static*. Entretanto, em protocolos projetados para redes ad hoc, o que normalmente se deseja verificar é o comportamento do protocolo em função da dinâmica da rede. Ou seja, a movimentação passa a ser o cenário sobre o qual o protocolo é verificado, e não parte da especificação deste. Nestes casos utilizaremos o parâmetro *dynamic*. Em termos da arquitetura, a utilização do parâmetro *dynamic* indicará a geração de um cenário dinâmico para a verificação do protocolo, o que não ocorrerá com o parâmetro *static*. Esta geração segue o modelo de mobilidade aleatório (*Random Walk Mobility Model*) (CBD02), onde a posição e a velocidade de um elemento é escolhida de forma aleatória. Utilizando qualquer um dos parâmetros, um nó pode verificar suas conexões diretas através do pronome *neighbours*, que será descrito na seção 3.2.3.

Estes parâmetros da topologia fazem referência a rede como um todo. Entretanto, é comum encontrarmos comportamentos distintos numa mesma rede. Por exemplo, a estabilidade da conexão entre estações base e estações móveis numa rede estruturada. Como trabalho futuro permitiremos que arestas específicas sejam classificadas com um ou outro parâmetro. Além disso, como já comentado, serão incorporadas construções probabilísticas. Com isso, será possível definir com que probabilidade uma determinada conexão existirá (ou falhará).

Em LEP temos macros para as topologias iniciais em anel (*Ring(N)*), estrela (*Star(C,N)*), sequência (*Seq(X)*), arbitrária (*Arbitrary(X)*) e totalmente conectada (*Complete(X)*). Os parâmetros N definem a dimensão da rede a ser gerada e C define o elemento central da estrela. Sempre que utilizamos uma especificação como a 3.3.c com uma topologia pré-definida ou combinada com uma especificação 3.3.b para definição de uma nova topologia, a arquitetura gera instâncias da topologia na sintaxe da especificação 3.3.a, de forma que essa

possa ser interpretada pelo módulo que processa uma especificação em LEP. Com esta possibilidade de geração várias instâncias (redes) de uma topologia, passamos a ter mais de um modelo a ser verificado. Ou seja, se o verificador retorna verdade para a verificação de uma propriedade numa especificação, significa que esta propriedade é válida para todas as instâncias da topologia descritas na especificação.

Especificações em Gramáticas de Grafos com Atributos

Nesta seção apresentamos as especificações de algumas das topologias pré-definidas em gramática de grafos com atributos de forma a ilustrar sua especificação. A definição de atributos é o mecanismo para a criação de novos pronomes, os quais poderão ser utilizados na especificação em LEP. A descrição sintática desta gramática é dada no Apêndice B.

Inicialmente apresentamos a especificação de uma topologia totalmente conectada na notação de gramática de grafos com atributos. Para exemplificar a utilização de atributos, utilizaremos dois (2) atributos (*s-neigh* e *h-neigh*), que definirão a vizinhança dos nós na rede. Em termos de gramática de atributos, *s-neigh* é um atributo sintetizado e *h-neigh* é um atributo herdado. A vizinhança de um nó será dada pelo seu atributo sintetizado *s-neigh*.

A notação gráfica da gramática apresentada na figura 3.4, para uma topologia totalmente conectada, se baseia na notação apresentada em (CGP00). Nesta, os nós com letra minúscula são os terminais e os com letra maiúscula são os não-terminais, os nós com círculo duplo são nós de saída e os que tem uma seta entrante são nós de entrada.

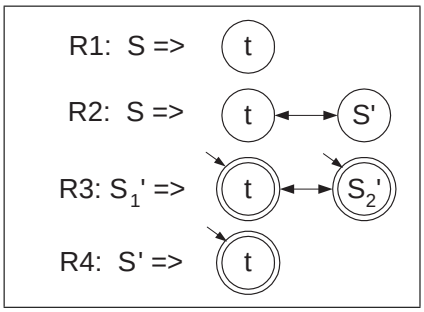

Figura 3.4: Gramática de Grafos para a topologia totalmente conectada

A notação textual em LEP (já com os atributos) equivalente à gramática apresentada na figura 3.4 é descrita abaixo.

S ⇒ t
S ⇒ t { t.s-neigh ← S'.s-neigh } ↔ S' { S'.h-neigh ← t }
S'$_1$ { S'$_1$.s-neigh ← S'$_1$.s-neigh ∪ {t} } ⇒
    in(out(t)) { t.s-neigh ← S'$_1$.h-neigh∪S'$_2$.s-neigh } ↔
    in(out(S'$_2$)) { S'$_2$.h-neigh ← t }
S' { S'.s-neigh ← S'.s-neigh ∪ {t} } ⇒
    in(out(t)) { t.s-neigh ← S'.h-neigh }

Para ilustrar a utilização da gramática graficamente, na figura 3.5 apresentamos a derivação da rede com 3 nós. Em cinza temos os não-terminais que são substituídos.

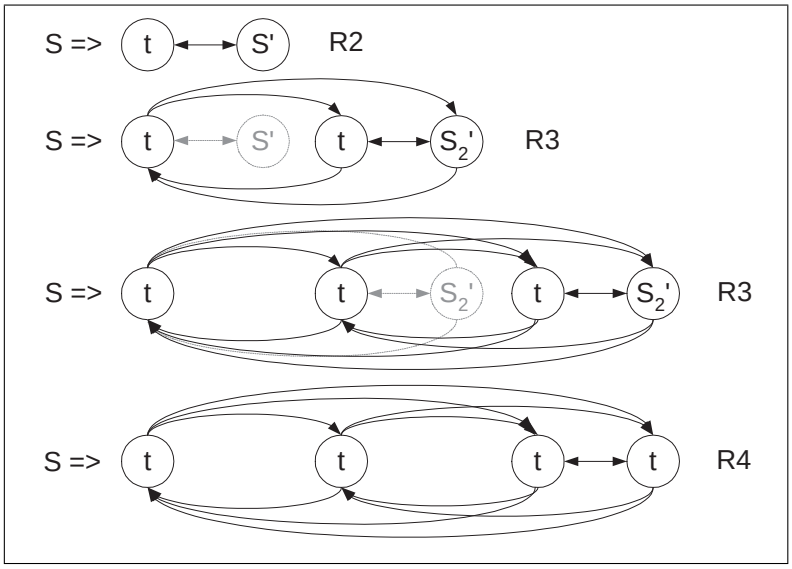

Figura 3.5: Derivação da Gramática de Grafos para a topologia totalmente conectada

Abaixo temos a especificação da topologia em anel, também com a definição da vizinhança:

S ⇒
    t { *t.s-neigh* ← *S'.s-neigh* } ↔ **S'** { *S'.h-neigh* ← *t* }
S'$_1$ { *S'*$_1$*.s-neigh* ← *t* } ⇒
    in(t) { *t.s-neigh* ← *S'*$_2$*.s-neigh* } →
    out(S'$_2$**)** { *S'*$_2$*.h-neigh* ← *S'*$_1$*.h-neigh* }
S' { *S'.s-neigh* ← *t* } ⇒
    in(out(t)) { *t.s-neigh* ← *S'.h-neigh* }

Como já salientamos anteriormente, LEP contém macros para as principais topologias. Ou seja, o usuário só deverá se preocupar com os detalhes desta especificação caso deseje alguma topologia não disponível.

3.2.3 Especificação dos Módulos

Um módulo ou conjunto de módulos em LEP definem o comportamento essencial de um protocolo ou algoritmo distribuído.

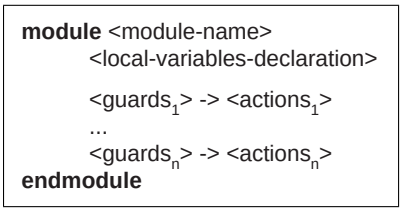

Figura 3.6: Declaração de um módulo em LEP

Um módulo (figura 3.6) consiste de um conjunto de variáveis locais e um conjunto de transições. Cada transição é composta de uma pré-condição (uma expressão booleana ou um comando de recebimento) e uma sequência de ações (possivelmente vazia)[3] que são executadas quando uma pré-condição é habilitada.

Uma pré-condição pode ser ainda as palavras reservadas *init*, *true* e *else*. A palavra *init* indica quais ações serão executadas no estado inicial do módulo. Quando uma ação pode ser executada sempre que possível, de forma não-determinística, utilizamos como pré-condição a palavra *true*. Finalmente, a ação correspondente à pré-condição *else* é executada sempre que nenhuma das pré-condições existentes é satisfeita. Quando mais de uma condição é satisfeita, também é feita uma escolha não-determinística entre elas. A execução de um processo (instância de um módulo) é uma repetição indefinida sobre as transições, excetuando a transição da pré-condição *init* que só é executada uma única vez.

As ações podem ser comandos de atribuição, sincronização (envio e recebimento), condicionais e repetições. A sincronização entre processos é feita através dos operadores de envio ("!") e recebimento ("?"). Além destes, os comandos *start* e *stop* iniciam ou terminam um ou mais processos. Outras estruturas sintáticas comuns como atribuições, condicionais, repetições são apresentadas através de exemplos na seção 3.2.5 ou na gramática de LEP em BNF descrita no Apêndice B.

Pronomes

Pronomes em LEP podem aparecer em qualquer posição na linguagem onde os nomes (identificadores dos elementos de um protocolo) ocorrem. Por

[3]Uma ação de uma transição pode ser vazia de maneira a permitir a verificação de especificações incompletas.

exemplo, podemos tê-los iniciando uma mensagem de envio, aguardando um recebimento ou passando-os como argumentos numa mensagem. Consideramos os seguintes:

this: pronome que faz referência ao próprio elemento onde o pronome ocorre; quando usado como um valor no lado direito de uma expressão, faz referência a um valor interno (não-visível ao usuário) que identifica este elemento;

sender: refere-se, no recebimento de uma mensagem, ao remetente da mensagem; útil na troca de mensagens do tipo *request-reply*; quando a mensagem recebida foi enviada para todos os elementos da rede através do pronome *everyone*, o uso deste pronome fará com que a resposta trafegue até alcançar quem originou a mensagem;

any(k, t): este é um pronome geral e parametrizado que refere-se a quaisquer k elementos do sistema de tipo t; este pronome insere não-determinismo no modelo;

anyother(x, k, t): este pronome refere-se a todos elementos retornados pelo pronome *any(k, t)*, exceto o pronome *this* no contexto onde o pronome *anyother(x, k, t)* ocorre;

everyone(t): utilizado num comando de envio, refere-se a todos os elementos do tipo t que são alcançados a partir de quem o chama; num comando de recebimento, refere-se a todos os elementos da rede, alcançáveis ou não; este pronome é implementado através de *flooding*[4] da mensagem requerida;

neighbours(k, t): este pronome refere-se ao conjunto de vizinhos do tipo t que estejam distantes k arestas do elemento onde este pronome ocorre;

none: não se refere a qualquer elemento na rede; tem a semântica de um valor nulo, ou seja, é útil para testarmos se algum outro pronome tem referências ou não;

parent: refere-se ao criador do elemento onde este pronome ocorre, o que independe se estes estão conectados diretamente no momento em que o pronome é executado;

children: refere-se a todos os elementos criados pelo elemento onde o pronome ocorre, o que também independe se estes estão conectados;

O parâmetro k dos pronomes, quando omitido, tem o valor padrão igual a um (1). O parâmetro t também é opcional e define os tipos de elementos (identificadores dos módulos) referenciados pelo pronome. Quando omitido, t tem o tipo do módulo corrente.

[4]Um algoritmo *flooding* distribui um dado para toda uma rede conexa por inundação

Além destes pronomes pré-definidos, podemos definir pronomes do usuário utilizando a definição de Gramática de Grafos com Atributos comentada na seção 3.2.2. Neste caso, sempre que o usuário definir uma gramática para uma topologia própria e definir atributos através dela, estes passam a ser pronomes a serem usados na especificação.

3.2.4 Especificação das Propriedades

Normalmente as lógicas para definição de propriedades sobre sistemas exigem bastante empenho por parte do testador. Procuramos amenizar este esforço da utilização de pronomes que mapeiam as propriedades padrões de sistemas concorrentes e reativos descritos em (DAC99). Por exemplo, a propriedade P_1

$$P_1\text{: [] (none!error)}$$

verifica se nunca a mensagem *error* é enviada. Utilizando os padrões propostos em (DAC99), este seria o padrão *Absence*, que é definido através do pronome *none*. Consideramos os seguintes pronomes de consulta:

none: este não se refere a qualquer elemento; em termos de propriedades, significa que nenhum nó se encontra num determinado estado;

everyone: este pronome representa um estado em que todos os nós de uma rede são referenciados;

any(k): representa o mesmo que o pronome de consulta *everyone* para k elementos da rede.

Além de pronomes, propriedades podem ser compostas de expressões booleanas, comandos de envio (*agente!msg(objeto)* - agente envia mensagem para objeto), comandos de recebimento (*objeto?msg(agente)* - objeto recebe mensagem de agente) e variáveis. Quando um identificador que ocorre numa propriedade não é um pronome pré-definido, este é uma variável e unifica com todas as ocorrências de variáveis com o mesmo nome na propriedade, ou seja, referenciam o mesmo nó. Por exemplo, a propriedade P_2

$$P_2\text{: [] (p!alive(everyone)}\rightarrow \text{<>p?ack(any(k)))}$$

verifica se sempre que um nó qualquer p envia uma mensagem *alive* para *everyone*, então em algum momento futuro o mesmo p receberá pelo menos k mensagens do tipo *ack*.

Alguns padrões listados em (DAC99) não são mapeados diretamente para LEP como foi o caso do padrão *Absence*. Um exemplo é o padrão *Response*, exemplificado na propriedade P_2. Nesta, apesar de também usarmos pronomes,

os operadores lógicos do padrão foram mantidos. Entretanto, de forma a aproximar mais ainda estas propriedades da linguagem natural, poderíamos combinar estes padrões, os pronomes e PSL (Property Specification Language) (Acc04), que é uma linguagem para especificação de propriedades que, dentre outras características, define termos em linguagem natural em substituição a operadores temporais. Assim, P_2 podería ser reescrita desta forma:

P_2': Always (p!alive(everyone)→ eventually p?ack(any(k)))

Para complementar, poderíamos até substituir o operador → pelo termo *implies*. Entretanto esta combinação está fora do escopo da tese, mas será comentada no capítulo 7.

3.2.5 Exemplos de especificações

Nesta seção apresentaremos alguns exemplos que ilustram o uso da linguagem LEP. Em **negrito** teremos as palavras reservadas de LEP e em *itálico* os pronomes. Primeiramente, a figura 3.7 mostra como um algoritmo de eleição de um líder poderia ser especificado para uma rede arbitrária.

```
topology is {1 - {2,3}, 2 - {1,3,4}, 3 - {1,2,4}, 4 - {2,3}} reliable;
module candidate
        vars my, p, count : int;
        init -> count = 0; my = this; neighbours!msg(my, count);
        this?win -> stop;
        this?msg(p, count) ->
                if ((p > my) or
                    ((p == my) and (count < topology.size))) then
                        my = p;  count = count + 1;
                        neighbours!msg(my, count);
                else
                        if ((p == this) and (count > topology.size)) then
                                everyone!win; stop;
                        endif
                endif
endmodule
```

Figura 3.7: Eleição de um líder numa rede arbitrária em LEP

Na figura 3.7 temos as definições da topologia e do módulo que especifica o algoritmo de eleição de um líder em LEP. Se substituíssemos a definição explícita da topologia por "*topology is Ring(<5) direct reliable*"que refere-se a topologias do tipo anel com até 4 nós, a especificação ainda sim funcionaria. Ou seja, podemos dizer que a separação sugerida por LEP entre a especificação da essência do protocolo e a especificação da topologia permite o reuso de especificações.

Como propriedade a ser verificada, podemos ter:

$$[]\ (\text{any!msg} \rightarrow <>\ \text{someone!win})$$

, que verifica se sempre que uma mensagem *msg* é enviada em algum momento futuro, a mensagem *win* também é enviada, ou seja, um líder é eleito. Como não há nenhum pronome com nome *someone*, este é uma variável na propriedade.

```
topology is Star(commander:1,soldier:1..6);
module commander
        init -> everyone!agree;
        this?yes(any(3)) -> everyone!consensus;
        this?no -> everyone!cancel;
endmodule
module soldier
        this?agree -> sender!yes;
        this?agree -> sender!no;
        this?cancel ->
        this?consensus ->
endmodule
```

Figura 3.8: Algoritmo de consenso especificado em LEP

Figura 3.8 apresenta a estrutura básica de um algoritmo de consenso, como um ataque coordenado com 1 comandante e no máximo 6 soldados, especificado em LEP. Nesta especificação, o comandante inicialmente envia uma mensagem *agree* para todos os soldados alcançáveis por ele. Depois, espera por no mínimo 3 mensagens concordando com o ataque (*yes*), respondendo com a mensagem para atacar (*consensus*), ou espera por uma mensagem negativa (*no*), e em seguida cancela a tentativa de ataque enviando a mensagem *cancel*. Os soldados por sua vez, esperam pelo ataque (mensagem *agree*), respondendo não-deterministicamente com as mensagens *yes* ou *no*, e esperam por mensagens que cancelam ou sinalizam o ataque (*cancel* e *consensus*, respectivamente), não enviando qualquer resposta no momento.

Quanto ao uso do pronome parametrizado *any(3)* na pré-condição *this?yes(any(3))*, a espera por mensagens do tipo *yes* funciona de forma não-bloqueante. Ou seja, a chegada das mensagens do tipo *yes* podem ser intercaladas com outras mensagens. A pré-condição *this?yes(any(3))* só é satisfeita após a chegada de três (3) mensagens do tipo *yes*.

Para esta especificação podemos testar as seguintes propriedades:

$$<>\ \text{anyone?no}$$

, que verifica se uma mensagem *no* eventualmente é necessariamente recebida;

$$[]\ \text{anyone?yes(everyone)}$$

, que verifica se um nó (o comandante nesta especificação) recebe uma mensagem *yes* de todo mundo;

$$[]\ \text{anyone?yes(any(3))}$$

, que é uma propriedade menos restrita que a anterior. Esta testa se pelo menos 3 mensagens *yes* são recebidas pelo comandante.

Na figura 3.9 apresentamos uma versão simplificada do protocolo DSR em LEP. A descrição completa deste protocolo é apresentada na seção 4.1.2.

```
topology is Arbitrary(5);
module x
        seq:x pacote;
        init -> pacote.first = this;
                pacote.last = anyother(this);
                neighbours!rr(pacote);
        this?rr (pacote) ->
                if this == pacote.last then
                        pacote.previous!unicast(pacote);
                else
                        pacote.add(this);
                        neighbours!rr(pacote);
                endif
        this?unicast (pacote) ->
                if this == pacote.first then
                        pacote.next!comm(pacote);
                else
                        pacote.previous!unicast(pacote);
                endif
        this?comm (pacote) ->
                if pacote.last <> this then
                        if neighbours.contains(pacote.next) then
                                pacote.next!comm(pacote);
                        else
                                neighbours!rr(pacote);
                        endif
                endif
endmodule
```

Figura 3.9: Especificação simplificada do DSR em LEP

Para este protocolo podemos ter a seguinte propriedade:

$$[]\ (\text{anyone!rr} \rightarrow <>\ \text{someone!comm})$$

, que verifica se sempre que uma requisição de rota é feita (mensagem *rr*), eventualmente uma mensagem *comm* é necessariamente enviada, indicando que uma rota foi calculada.

3.2.6 Modelo de Comunicação

Nesta seção apresentamos o modelo de comunicação de LEP. Para a especificação desse modelo em SOS (Plo81) utilizamos uma estrutura que chamamos de ambiente (figure 3.10). O ambiente é um grafo onde nós v_i representam os processos (instâncias de um módulo) e arestas a_{ij} representam as conexões existentes entre os processos. Cada nó é associado a um elemento lógico que contém buffers de mensagens e um conjunto de variáveis locais que determinam seu estado interno.

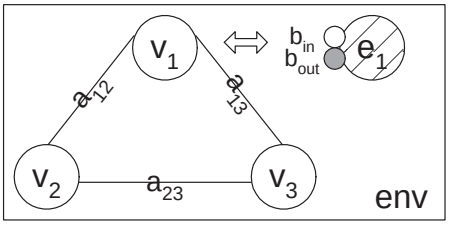

Figura 3.10: Ambiente do Modelo de Comunicação de LEP

No envio de uma mensagem, o processo remetente coloca a mensagem em seu buffer de saída para que esta possa ser passada para o buffer de entrada do processo destino. O tamanho do buffer e a maneira como as mensagens são armazenadas nele (FIFO, LIFO, conjunto, ..) determinam o comportamento da rede. Por exemplo, da forma usual, se o tamanho do buffer é zero (0), temos uma comunicação do tipo *rendezvous* ou, se o buffer é manipulado como um conjunto, não garantimos ordem na entrega de mensagens. Quanto ao buffer de saída, este é interessante em situações como envio de mensagens num canal não-bloqueante onde ocorre uma desconexão.

Além da comunicação externa entre processos, cada processo tem transições internas que podem mudar seu estado. Como vimos, o estado de um processo é composto do seu conjunto de variáveis locais juntamente com o status dos buffers. Estes estados serão utilizados na verificação de propriedades sobre o sistema.

Este modelo de computação está associado à linguagem intermediária da arquitetura, que tem por objetivo unificar os conceitos de protocolos e sistemas distribuídos. Esta linguagem define estes protocolos no mesmo nível que as linguagens dos projetos (BGM02, KG02, JHA+96) pois, comparando com LEP, nestas linguagem a comunicação entre os seus elementos se dá de maneira singular (1x1) e a topologia de suas especificações é fornecida "amarrada"à definição do protocolo. Ou seja, para outra topologia, devemos alterar a especificação do protocolo como um todo. A definição em linhas gerais desta linguagem é dada na seção 3.3.1.

Para a definição do modelo de comunicação em SOS, consideramos as seguintes categorias sintáticas:

$e \in Elem = \{Id \times Buf \times Buf \times Loc \times Trans\}$ = Processos
$v \in Vrt$ = Conjunto de vértices
$a \in Edge = \{Vrt \times Vrt\}$ = Arestas
$gr \in Grf = Vrt \cup Edge$ = Grafo
$ch \in Chan$ = Conjunto de canais
$env \in Env$ = Ambiente

Utilizamos essa notação abstrata dos processos como uma quíntupla de forma a não poluir demasiadamente a descrição do modelo de comunicação (Na implementação um processo é um módulo no código intermediário). Um processo se constitui de um identificador (*Id*), dois (2) buffers para entrada e saída de mensagens (*Buf*), uma memória local (*Loc*) e um conjunto de transições (*Trans*). Neste modelo apresentamos apenas as trocas de mensagens e movimentações, as quais são percebidas externamente aos processos. Os comandos executados internamente a cada processo (repetições, condicionais, atribuições, ..) são transições silenciosas neste modelo.

A sintaxe abstrata do modelo de comunicação de LEP é descrita à seguir:

$$\frac{v : Vrt}{v : Grf} \qquad \frac{v_1 : Vrt \;\; v_2 : Vrt \;\; a : Aresta}{v_1 \xrightarrow{a} v_2 : Grf}$$

$$\frac{gr_1 : Grf \;\; gr_2 : Grf}{gr_1 \;\; gr_2 : Grf} \qquad \frac{gr : Grf \;\; f : Vrt(gr) \rightarrow Elem \;\; g : Chan \rightarrow Id}{< gr, f, g >: Env}$$

Uma rede (*Grf*) pode ser formada por um único nó (*Vrt*), pela junção de subgrafos com ou sem conexão (*Aresta*). Um ambiente (*Env*) é composto pela rede, uma função f que mapeia cada nó um elemento da especificação (instância, agente do protocolo) e uma função g que define, para um dado canal, a que elemento este pertence.

As regras semânticas do modelo de comunicação são:

Comportamento:

$$\frac{Seja: \;\; A =< id_a, b_{in}, b_{out}, assoc, t >, \;\; A\prime =< id_a, b_{in}\prime, b_{out}\prime, assoc\prime, t\prime > \quad A \rightarrow A\prime}{< gr, f, g > \rightarrow < gr, f\prime, g >}$$
, *onde*: $f(v) = A, f\prime(v) = A\prime, \{\forall v_1, v_1 \neq v, f(v_1) = f\prime(v_1)\}$

A evolução de um elemento é refletida no ambiente pela função *f*, que mapeia cada nó num elemento.

Transições internas:

$$
\begin{array}{l}
< id, b_{in}, b_{out}, assoc_{local}, t > \rightarrow \\
\qquad < id, b_{in}, b_{out}, assoc'_{local}, t > \\
\\
< id, b_{in}, b_{out}, assoc_{local}, ch!m(val)|t > \rightarrow \\
\qquad < id, b_{in}, b_{out}.ch!m(val), assoc_{local}, t > \\
\\
< id, ch?m(val)|b_{in}, b_{out}, assoc_{local}, t[x] > \rightarrow \\
\qquad < id, b_{in}, b_{out}, assoc_{local}, t[m(val)/x] >
\end{array}
$$

Uma transição interna num elemento da rede pode ser: a execução de uma transição, o que pode causar a alteração em sua memória local; inserção de uma mensagem no buffer de saída para envio, e; processamento de uma mensagem armazenada no buffer de entrada.

Nestas regras, o operador '|' separa o conjunto de mensagens em primeiro e restante, '.' insere uma mensagem num buffer e o termo *t[m(val)/x]* é uma espécie de λ-abstração que evoluirá o estado do elemento levando-se em conta que a mensagem *m(val)* foi recebida.

Regra de sincronização:

$$
\begin{array}{l}
Seja: \\
A =< id_a, b_{in}, ch!m(val)|b_{out}, l_{local_1}, t >; \\
A\prime =< id_{a\prime}, b_{in}, b_{out}, l_{local_1}, t >; \\
B =< id_b, ch?x|b_{in_2}, b_{out_2}, l_{local_2}, t_2 >; \\
B\prime =< id_{b\prime}, ch?m(val)|b_{in_2}, b_{out_2}, l_{local_2}, t_2 > \\
\qquad A \rightarrow A\prime \;\; B \rightarrow B\prime \\
\hline
< gr, f, g > \rightarrow < gr, f\prime, g > \\
, \; onde: \; v_1, v_2 \in gr, \; f(v_1) = A, \; f(v_2) = B, \; v_1 \xrightarrow{a} v_2 \in gr, \; g(ch) = id_b
\end{array}
$$

Uma sincronização é realizada quando o remetente prepara uma mensagem em seu buffer de saída e o buffer de entrada do destinatário aguarda por esta mensagem.

Inserção de aresta:

$$
\begin{array}{l}
\dfrac{< id, b_{in}, b_{out}, assoc_{local}, insert(a, v_2)|t > \rightarrow < id, b_{in}, b_{out}, assoc_{local}, t >}{< gr, f, g > \rightarrow < gr\prime, f, g >} \\
, \; onde: \; \{v_1, v_2\} \subset gr, \; f(v_1) =< id, b_{in}, b_{out}, assoc_{local}, insert(a)|t >, \\
v_1 \xrightarrow{a} v_2 \notin gr, \; gr\prime = gr \cup \{v_1 \xrightarrow{a} v_2\}
\end{array}
$$

Remoção de aresta:

$$\frac{< id, b_{in}, b_{out}, assoc_{local}, remove(a)|t > \rightarrow < id, b_{in}, b_{out}, assoc_{local}, t >}{< gr, f, g > \rightarrow < gr\prime, f, g >}$$
, *onde* : $\{v_1, v_2\} \subset gr,\ f(v_1) =< id, b_{in}, b_{out}, assoc_{local}, remove(a)|t >,$
$v_1 \xrightarrow{a} v_2 \in gr,\ gr\prime = gr - \{v_1 \xrightarrow{a} v_2\}$

As inserções e remoções de arestas referem-se às movimentações dos elementos de rede.

Inserção de vértice:

$$\frac{< id, b_{in}, b_{out}, assoc_{local}, insert(v_1)|t > \rightarrow < id, b_{in}, b_{out}, assoc_{local}, t >}{< gr, f, g > \rightarrow < gr\prime, f\prime, g >}$$
, *onde* :
$f(v) =< id, b_{in}, b_{out}, assoc_{local}, insert(v_1)|t >,\ v \in gr,$
$\nexists e, f(v_1) = e,\ v_1 \notin gr,\ f\prime(v_1) = e,\ e \in Elem,$
$gr\prime = gr \cup \{v_1\}$

Remoção de vértice:

$$\frac{< id, b_{in}, b_{out}, assoc_{local}, remove(v_1)|t > \rightarrow < id, b_{in}, b_{out}, assoc_{local}, t >}{< gr, f, g > \rightarrow < gr\prime, f\prime, g >}$$
, *onde* :
$\{v, v_1\} \subset gr,\ v \xrightarrow{a} v_1 \in gr,$
$f(v) =< id, b_{in}, b_{out}, assoc_{local}, remove(v_1)|t >,$
$f\prime(v_1) = \phi,\ gr\prime = gr - \{v \xrightarrow{a} v_1\}$

As inserções e remoções de vértices referem-se as possíveis conexões e desconexões dos elementos na rede.

3.3 Semântica Formal das Traduções

O objetivo desta seção é dar uma noção mais construtiva de como são feitas as traduções entre os módulos da arquitetura. Descreveremos as traduções de LEP para o Código Intermediário, do Código Intermediário para Promela e SMV. As traduções serão apresentadas como regras de reescrita ($x \triangleright y$ representa a reescrita de x na linguagem origem para y na linguagem destino).

3.3.1 LEP para o Código Intermediário

Nesta seção apresentamos uma descrição formal de como especificações descritas em LEP são traduzidas para o Código Intermediário. Na tradução, mapeamos os pronomes para seus respectivos significados, baseado na topologia e no contexto (local onde o pronome é utilizado) dados. As trocas de mensagens são especificadas apenas no Código Intermediário, já que é mais simples especificar uma comunicação um para um do que um para muitos feita pelos pronomes.

Considerando as categorias sintáticas listadas na seção 3.2.6, adicionamos as seguintes:

mid ∈ MId = Conjunto de identificadores de módulos
mod ∈ Mod = Conjunto dos módulos
t ∈ Trans = CExp × Cmd = Conjunto de transições
ce ∈ CExp = Conjunto de expressões condicionais
m ∈ Msg = Conjunto de mensagens
cm ∈ Cmd = Conjunto de comandos
ch ∈ Chan = Conjunto de canais
top ∈ Top = Conjunto de topologias
bool ∈ Bool = Conjunto de valores booleanos
connec ∈ MId → MId = Conjunto de conexões
prop ∈ Prop = Conjunto das propriedades

No modelo semântico, *top* é a topologia da rede, *st* é o estado da tradução, que contém o identificador do módulo corrente e a topologia. As funções f:Pron × MId × Top→{Chan} e g:Chan→ MId retornam informações sobre a topologia da rede. Mensagens em LEP são traduzidas para mensagens no Código Intermediário com dois (2) argumentos adicionais que podem ser vistos como rótulos das mensagens: o identificador do remetente da mensagem e um valor booleano que indica se a mensagem precisa ser reencaminhada (como na implementação do pronome *everyone*). As regras semânticas são (especificações entre ⟦⟧ representam elementos a serem avaliados pelo processo de reescrita):

Tradução da especificação:

<*topology is connections params; mod prop*> ▷
⟦*mod*, ⟦processTopology(connections,params,mod)⟧⟧

Uma especificação em LEP, como vimos, é composta da expressão que define a topologia, da definição dos módulos que compõem o protocolo e da

propriedade a ser verificada. A função *processTopology* fornece informação sobre a topologia para ser usada na tradução.

$$
\begin{array}{|l}
\hline
<\mathit{module\ mid\ t\ endmodule}> \triangleright \\
\quad < b_{in}, b_{out}, assoc, [\![t, st(mid, top)]\!] >, \\
\mathit{onde}: \ |b_{in}| = |b_{out}| = 0, \{\forall v \ assoc(v) = 0\} \\
\hline
\end{array}
$$

Cada módulo em LEP é transformado em um elemento no modelo de comunicação. O termo *(*st) (*store*) representa uma memória temporária para o processo de reescrita. Dentre outras funções, esta armazena o contexto (módulo corrente) de uma dada avaliação (*mid*).

Tradução das transições:

$$
\begin{array}{|l}
\hline
< t_{lep_1}\ t_{lep_2}, st(mid, top) > \triangleright \\
\quad [\![t_{lep_1}, st(mid, top)]\!][\![t_{lep_2}, st(mid, top)]\!] \\
\hline
\end{array}
$$

Tradução de cada transição:

$< ce_{lep} \rightarrow cm_{lep}, st(mid, top) > \triangleright$
$\quad [\![ce_{lep}, st(mid, top)]\!]$ ->
$\quad\quad [\![\text{generateConditionEveryone}(ce_{lep}, mid, top)]\!]$
$\quad\quad [\![cm_{lep}, st(mid, top)]\!]$

Para o caso em que a rede é *unreliable*, ou seja, mensagens podem ser descartadas:
$< ce_{lep} \rightarrow cm_{lep}, st(mid, top) > \triangleright$
$\quad [\![ce_{lep}, st(mid, top)]\!]$ ->
$\quad\quad [\![\text{generateConditionEveryone}(ce_{lep}, mid, top)]\!]$
$\quad\quad [\![cm_{lep}, st(mid, top)]\!]$
$\quad [\![ce_{lep}, st(mid, top)]\!]$->

Quando a rede é *unsecure*, ou seja, valores podem ser corrompidos:
$< ce_{lep} \rightarrow cm_{lep}, st(mid, top) > \triangleright$
$\quad [\![ce_{lep}, st(mid, top)]\!]$ ->
$\quad\quad [\![\text{generateConditionEveryone}(ce_{lep}, mid, top)]\!]$
$\quad\quad [\![cm_{lep}, st(mid, top)]\!]$
$\quad not\ [\![ce_{lep}, st(mid, top)]\!]$ ->
$\quad\quad [\![\text{generateConditionEveryone}(ce_{lep}, mid, top)]\!]$
$\quad\quad [\![cm_{lep}, st(mid, top)]\!]$

Como exemplo de implementação dos parâmetros da topologia, temos a tradução de uma transição para os casos em que a topologia é a padrão, *unreliable* ou *unsecure*. A função *generateConditionEveryone* gera comandos que verificam se a mensagem recebida precisa ser reencaminhada ou se já foi recebida e deve ser descartada, para o caso de mensagens enviadas com o pronome *everyone*.

Tradução do pronome this numa pré-condição de uma transição:

<this?m, st(mid,top)> ▷ ch?m, onde: g(ch)=mid

Tradução do pronome this no lado direito de uma expressão:

<this, st(mid,top)> ▷ mid

Tradução de uma sequência de comandos:

$$
\begin{array}{l}
< cm_{lep_1}; cm_{lep_2}, st(mid, top) > \triangleright \\
\quad [\![cm_{lep_1}, st(mid, top)]\!] \\
\quad ; \\
\quad [\![cm_{lep_2}, st(mid, top)]\!]
\end{array}
$$

Tradução de um comando condicional:

$$
\begin{array}{l}
< if\ ce\ then\ cm\ endif, st(mid, top) > \triangleright \\
\quad if\ [\![ce, st(mid, top)]\!]\ \{ \\
\quad\quad [\![cm, st(mid, top)]\!] \\
\quad \}
\end{array}
$$

Tradução do pronome neighbours num comando de envio:

$$
\begin{array}{l}
< neighbours!m, st(mid, top) > \triangleright \\
\quad local_1 = 0; \\
\quad while\ (local_1 <= k)\ \{ \\
\quad\quad ch[local_1]!m(mid, false); \\
\quad\quad local_1 + +; \\
\quad \} \\
,\ onde : \text{processPronoun}(neighbours, mid, top) = \{ch[1], ..., ch[k]\}, \\
g(ch[1]) = mid_1, g(ch[k]) = mid_k, \{mid_1, mid_k\} \cup MId, \\
k = |\text{processPronoun}(neighbours, mid, top)|
\end{array}
$$

Na tradução do pronome *neighbours*, a função *processPronoun*, que recebe como entrada um dado pronome, um elemento na rede e uma topologia, retorna o conjunto de canais associados a este pronome. No código intermediário temos uma iteração nos canais retornados, inserindo em cada um a mensagem encaminhada pelo pronome. No comando de envio $ch[local_1]!m(mid, false);$, o parâmetro *false* indica que a mensagem não será enviada por inundação (*flooding*), como ocorre com o pronome *everyone*.

3.3.2 Código Intermediário para Promela

Nesta seção apresentamos como a representação intermediária é mapeada em Promela (linguagem de especificação do Spin) (Hol97). Esta tradução é mais direta que a anterior, a qual já substituiu os pronomes por suas referências equivalentes. Considerando também as categorias sintáticas das seções anteriores, temos as seguintes regras de reescrita:

Tradução do conjunto de elementos do Código Intermediário para Promela

$$
\begin{array}{|l|}
\hline
<< mid, b_{in}, b_{out}, assoc, t > \ e_2, top, prop > \rhd \\
\quad [\![\text{declare-channel-msgs}(mid, t)]\!] \\
\quad [\![\text{declare-vars}(prop)]\!] \\
\quad [\![\text{declare-runs}(mid, top)]\!] \\
\quad [\![< mid, b_{in}, b_{out}, assoc, t >, top, prop]\!] \\
\quad [\![e_2, top, prop]\!] \\
\hline
\end{array}
$$

A função *declare-channel-msgs* define os canais de mensagens e os tipos de mensagens trocadas pelos processos, que são necessários para comunicação destes em Promela. A função *declare-vars* declara variáveis globais relativas à propriedade a ser verificada. Por exemplo, se na propriedade ocorre uma subfórmula como $p!msg$, então uma variável global é declarada de forma que os envios desta mensagem possam ser registrados. A função *declare-runs* insere no procedimento principal de Promela a criação de um processo baseado neste módulo.

Tradução de cada nó para um processo em Promela:

$$
\begin{array}{l}
<< mid, b_{in}, b_{out}, assoc, t >, top, prop > \triangleright \\
proctype\ mid\ \{ \\
\quad [\![\text{declare-locals}(assoc)]\!] \\
\quad [\![\text{declare-init}(t)]\!] \\
\quad do \\
\quad\quad [\![t, top, prop]\!] \\
\quad od \\
\}
\end{array}
$$

A função *declare-locals* declara as variáveis locais de um processo. A função *declare-init* insere na especificação os comandos que são executados no início de cada processo.

Tradução das transições:

$$< t_1\ t_2, top, prop > \triangleright [\![t_1, top, prop]\!][\![t_2, top, prop]\!]$$

Tradução de cada transição:

$$
\begin{array}{l}
< ce \rightarrow cm, top, prop > \triangleright \\
\quad :: [\![ce]\!] \rightarrow [\![\text{update-vars}(ce, prop)]\!][\![cm, top, prop]\!]
\end{array}
$$

A função *update-vars* incrementa as variáveis globais criadas pela função *declare-vars* que serão usadas pelas propriedades para verificação em Spin.

Tradução de uma expressão condicional:

$$
\begin{array}{l}
< if\ ce\{cm\}, top, prop > \triangleright \\
\quad if \\
\quad\quad :: [\![ce]\!] \rightarrow [\![\text{update-vars}(ce, prop)]\!][\![cm, top, prop]\!] \\
\quad\quad :: else \\
\quad fi
\end{array}
$$

Tradução de um comando de envio:

$$
\begin{array}{l}
< ch!m, top, prop > \triangleright \\
\quad ch!m;\ [\![\text{update-vars}(ch!m, prop)]\!]
\end{array}
$$

3.3.3
Código Intermediário para SMV

Similarmente à seção 3.3.2, apresentamos nessa seção a tradução da representação intermediária para SMV (CCGR00). Esta tradução não é tão direta, já que a implementação básica de SMV não contém primitivas para comunicação entre processos. Além disso, nas transições de estado num módulo em SMV as atribuições das variáveis são feitas de maneira simultânea. Ou seja, precisamos sequenciar essas atribuições num mesmo módulo para que a tradução reflita a semântica do Código Intermediário.

Tradução do conjunto de elementos do Código Intermediário:

$$
\begin{array}{l}
< e_1\ e_2, top, prop > \triangleright \\
\quad MODULE\ main \\
\quad\quad VAR\ [\![\text{declare-globals}(e_1, e_2, top)]\!] \\
\quad\quad ASSIGN\ [\![\text{assign-globals}(e_1, e_2, top)]\!] \\
\quad\quad [\![e_1, top]\!][\![e_2, top]\!]
\end{array}
$$

Nestas regras de tradução, a função *declare-globals* declara instâncias (nós, processos) que compõe a rede e a matriz de conexões da rede. Como estamos modelando sistemas concorrentes, declaramos as instâncias como *process* em SMV. Cada instância recebe como parâmetro o seu identificador, as instâncias dos outros processos existentes e a matriz de conexões. Na troca de mensagens, o remetente vê na lista processos que este tem se o processo destinatário está disponível. O destinatário faz o mesmo teste para algum remetente. Estando ambos disponíveis, a comunicação se estabelece. A função *assign-globals* inicializa a matriz de conexões (matriz de booleanos).

Tradução de cada elemento para um módulo SMV:

$$
\begin{array}{l}
<< mid, b_{in}, b_{out}, assoc, t >, top > \triangleright \\
\quad MODULE\ \text{mod\_mid}(mid, processes, matrix) \\
\quad\quad VAR\ [\![\text{declare-locals}(assoc)]\!][\![\text{declare-states}(t)]\!] \\
\quad\quad ASSIGN\ [\![\text{assign-initial-states}(t)]\!][\![t, top]\!] \\
\quad\quad FAIRNESS\ running
\end{array}
$$

A função *declare-locals* declara variáveis locais como *partner* e *parent*, que armazenam os identificadores de um processo que esteja se comunicando com o processo corrente e do processo pai, respectivamente. Além destas, esta declara as variáveis locais da própria especificação, as quais serão alteradas durante as simulações. A função *declare-states* declara uma variável

que armazena os possíveis estados da instância, baseada nas transições que correspondem às arestas de entrada e saída do respectivo nó no Código Intermediário. A função *assign-initial-states* atribui o estado inicial das variáveis de acordo com sua natureza.

Tradução das transições:

$$\boxed{\begin{array}{l} < t_1\ t_2, top > \triangleright \\ \quad [\![t_1, top, 0]\!][\![t_2, top, 0]\!] \end{array}}$$

O valor zero (0) passado como parâmetro na avaliação das transições representa o valor inicial do parâmetro *pos*, o qual indica a posição do comando traduzido na sequência onde este ocorre. Esta informação é importante pois as atribuições das variáveis de um módulo em SMV são feitas de maneira simultânea. Ou seja, utilizaremos os valores de *pos* para forçar o sequenciamento da execução num mesmo módulo.

Tradução de uma transição com atribuição:

$$\boxed{\begin{array}{l} << ce \rightarrow id = expr >,\ cm, top, pos > \triangleright \\ \quad next(id) := \\ \qquad case \\ \qquad\quad order = pos\ \&\ [\![\text{generate-pre-conds}(ce, id, top)]\!]\ :\ [\![expr]\!]; \\ \qquad esac; \\ \quad [\![ce \rightarrow cm, top, inc(pos)]\!] \end{array}}$$

Finalmente, a função *generate-pre-conds* gera as pré-condições das transições de cada módulo do SMV à partir de uma transição na representação intermediária. Nas pré-condições utilizamos a variável *order*, a qual serve para executar os comandos da transição em sequência, uma vez que as associações de um módulo em SMV são todas feitas em paralelo. A variável *order* definirá a ordem de execução dos comandos na sequência que estes ocorrem na transição.

Tradução de uma transição com condicional:

$$\boxed{\begin{array}{l} << ce_1 \rightarrow if(ce_2)\{cm_1\} >,\ cm_2, top, pos > \triangleright \\ \quad [\![[\![ce_1]\!]\ \&\ [\![ce_2]\!] \rightarrow cm_1, top, pos]\!] \\ \quad [\![ce_1 \rightarrow cm_2, top, inc(pos)]\!] \end{array}}$$

3.3.4 Recuperação dos Contra-Exemplos com Pronomes

Nesta seção descrevemos de maneira geral os métodos utilizados para se obter os contra-exemplos retornados pelo Spin e pelo SMV no nível de abstração de LEP. O objetivo é simplificar o contra-exemplo retornado, de forma que este fique mais legível e seja compatível com o nível de abstração de LEP. Os passos em comum tomados por estes métodos podem ser generalizados para serem aproveitados na extensão da arquitetura para outros verificadores.

A recuperação dos contra-exemplos, assim como a especificação do protocolo ou algoritmo, é feita em dois (2) passos. Na ida (processo de geração da especificação), alimentamos a especificação com informações (marcadores) que auxiliarão na recuperação do contra-exemplo. Na volta (interpretação dos contra-exemplos, quando existem), detectamos no contra-exemplo as informações inseridas na primeira parte, as quais nos auxiliarão na recuperação. Novamente seguindo a filosofia da arquitetura, ambas as etapas são feitas de forma automática e transparente para o usuário.

Os contra-exemplos no nível de LEP são sequências de ações no formato $a_1 \Rightarrow a_2 \Rightarrow ... \Rightarrow a_k$, onde cada a_j é um comando de envio ou recebimento, cujos agentes ocorrem na propriedade que foi verificada. a_1 é a ação inicial do contra-exemplo e a_k é a ação final, onde a propriedade é invalidada. Como citado anteriormente, identificadores que não sejam palavras reservadas de LEP (pronomes) são variáveis, as quais unificam com variáveis que tenham o mesmo nome.

Na geração do Código Intermediário a ser feita pelo módulo *LEP2CI* da figura 3.2, inserimos os marcadores, que são identificadores únicos e servem para indicar uma posição na especificação em LEP durante a recuperação de um contra-exemplo. Neste caso, os marcadores indicam em que ponto uma dada propriedade foi invalidada na especificação em LEP, ou os caminhos percorridos até a propriedade ser invalidada.

De forma a simplificar o processamento, implementamos o marcador como uma variável (*codePositionLEP*) no domínio dos naturais. Esta variável é gerada na tradução da especificação de LEP para o Código Intermediário. Ela tem o valor inicial zero (0) e é incrementada no Código Intermediário sempre que um novo ponto de interesse ocorre em LEP. Estes pontos de interesse são: no início da especificação de cada módulo; no início do conjunto de transições, e; no início de uma ação numa transição, ou seja, após a respectiva pré-condição ser satisfeita.

Para exemplificarmos, na figura 3.11 apresentamos o exemplo do ataque coordenado discutido anteriormente.

```
topology is Star(commander:1,soldier:1..6);
module commander
        init -> everyone!agree;
        this?yes(any(3)) -> everyone!consensus;
        this?no -> everyone!cancel;
endmodule
module soldier
        this?agree -> sender!yes;
        this?agree -> sender!no;
        this?cancel ->
        this?consensus ->
endmodule
```

Figura 3.11: Algoritmo de consenso especificado em LEP

Para o módulo do comandante, um trecho do módulo equivalente gerado no Código Intermediário é apresentado na figura 3.12.

Ou seja, os marcadores nada mais são do que atribuições a variáveis. Como havíamos dito, esta implementação facilita as traduções, já que atribuições são construções básicas da maioria das linguagens de especificação.

Para gerarmos um contra-exemplo nos verificadores, utilizaremos a propriedade:

$$[]\ \text{none!cancel}$$

, a qual expressa que em nenhum momento alguém envia uma mensagem do tipo *cancel*. Pela especificação, obviamente este alguém é o comandante. Como os soldados respondem não-deterministicamente, em algum momento eles podem discordar do ataque. Logo, a propriedade não é válida para esse modelo.

No Código Intermediário, esta propriedade é escrita como:

$$[]\ (\text{not(commander[1]!cancel) and not(soldier[1]!cancel) and not(soldier[2]!cancel) and not(soldier[3]!cancel))}$$

, já que a pergunta foi feita de forma geral. Se especificássemos explicitamente o comandante, a expressão no Código Intermediário só faria referência à ele.

Tanto em Spin quanto em SMV, a recuperação do contra-exemplo no nível de LEP com respeito a uma propriedade tem um primeiro passo em que negamos esta propriedade. Para geração da negação da propriedade, além da negação usual dos operadores lógicos, também temos que inverter o significado dos pronomes, quando estes aparecem como agentes. A inversão dos pronomes que especificam propriedades é dada pela tabela 3.1.

```
codePositionLEP=0;
i=0;
while (i<quant_neighbours) {
        neighbours[i]!agree(commander[myid],true);
        i++;
}
while (true) {
        codePositionLEP=1 ;
        trans {
                commander[myid]?yes(sender,type) ->
                codePositionLEP=2;
                count_any1++;
                if (count_any1 == 3) {
                        codePositionLEP=3;
                        i=0;
                        while (i<quant_neighbours) {
                                neighbours[i]!consensus(commander[myid],true);
                                i++;
                        }
                }
                commander[myid]?no(sender,type) ->
                        codePositionLEP=4;
                        i=0;
                        while (i<quant_neighbours) {
                                neighbours[i]!cancel(commander[myid],true);
                                i++;
                        }
        }
}
```

Figura 3.12: Trecho gerado da tradução do módulo comandante para o Código Intermediário

none	any
everyone	not(any) / p
any	none
p	not(p)

Tabela 3.1: Negação dos pronomes como agentes

Spin

Nesta subseção apresentaremos como processamos as saídas do Spin para retornar os contra-exemplos no nível de abstração de LEP.

O módulo que faz a conversão do Código Intermediário para Promela (*CI2Spin* na figura 3.2) insere na especificação em Promela os marcadores e as variáveis que controlam o número de mensagens do tipo *cancel* enviadas por cada elemento (figura 3.13).

A propriedade reescrita em LTL é descrita como abaixo:

$$[]\ (\text{nr-cancel-commander}==0)\ \&\&\ (\text{nr-cancel-soldier1}==0)\ \&\&\ (\text{nr-cancel-soldier2}==0)\ \&\&\ (\text{nr-cancel-soldier3}==0)$$

Assim, o Spin retorna o contra-exemplo de forma gráfica (figura 3.14) e textual (figura 3.15), a qual é processada para podermos recuperar a especificação no nível inicial.

```
do
        :: codePositionLEP=1;
          if
          :: commander_c[myid]?yes(sender,every) ->
                codePositionLEP=2 ;
                count1++;
                if
                :: count1==3 ->
                        codePositionLEP=3;
                        i=0;
                        do
                        :: i<quant_neighbours ->
                                neighbours[i]!consensus(commander_c[myid],1);
                                i++
                        :: else -> break
                        od
                :: else ->
                fi
          :: commander_c[myid]?no(sender,every) ->
                codePositionLEP=4;
                i=0;
                do
                :: i<quant_neighbours ->
                        neighbours[i]!cancel(commander_c[myid],1);
                        nr_cancel_commander++;
                        i++
                :: else -> break
                od
          fi
od
```

Figura 3.13: Trecho do código da especificação do comandante em Promela

Da versão textual do contra-exemplo (figura 3.15) podemos identificar, para cada instrução executada, informações que auxiliam na recuperação dos contra-exemplos no nível de LEP, como: identificação do processo que executou a instrução; a qual módulo está associado este processo, e; a instrução que foi executada (por exemplo, as atribuições à variável *codePositionLEP*).

Para este cenário, o contra-exemplo no nível de LEP que queremos obter é:

```
commander!agree(everyone) =>
    commander?no =>
        commander!cancel(everyone)
```

, que é uma sequência de comandos existentes na especificação em LEP.

Para obtermos esta sequência, primeiramente negaremos a propriedade que foi invalidada. Após, buscaremos a sequência para a qual a propriedade negada é verdadeira, tendo como parâmetros: o código original em LEP, o formato textual do contra-exemplo e os marcadores. Esta busca é feita de trás para frente na linha do tempo. Ou seja, iniciamos pelo último estado do contra-exemplo, o qual invalida a propriedade original e, tendo a especificação em LEP como referência, identificamos a condição anterior para que este estado tenha sido alcançado. Fazemos isto sucessivamente até que não tenhamos mais pré-condições. Esta busca no sentido contrário é necessária pois nos contra-exemplos retornados pelo Spin podem existir ramos que não tenham influência

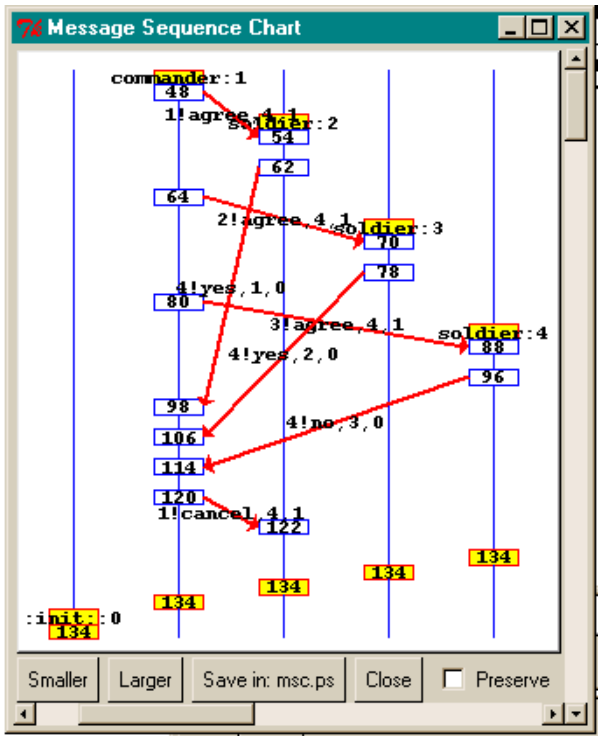

Figura 3.14: Contra-exemplo no formato MSC retornado pelo Spin

direta no contra-exemplo. Ou seja, se fizéssemos uma busca do início para o fim, poderíamos escolher algum ramo que não invalidasse a propriedade analisada.

O método de recuperação segue os seguintes passos:

1. Gerar a negação $\bar{\phi}_{Promela}$ da propriedade $\phi_{Promela}$;

2. Buscar no contra-exemplo textual o estado onde $\bar{\phi}_{Promela}$ é satisfeita; consequentemente teremos o comando que foi executado e o marcador corrente;

3. Dados o marcador e comando obtidos, buscar comando correspondente na especificação em LEP; este será o último elemento da sequência a ser retornada;

4. Também na especificação em LEP, buscamos o comando que seja uma pré-condição para a execução deste comando; necessariamente, o agente deste comando (para o caso de comandos de sincronismo) precisa ocorrer em ϕ;

5. Repetimos o passo anterior até não encontrarmos mais nenhuma pré-condição;

6. Ao final a sequência retornada é a já construída.

```
128:    proc  1 (commander) line  67 "pan_in" (state 41)    [codePositionLEP = 4]
<merge 49 now @42>
128:    proc  1 (commander) line  68 "pan_in" (state 42)    [i = 0]      <merge 49
now @49>
130:    proc  1 (commander) line  70 "pan_in" (state 43)
[((i<quant_neighbours))]
132:    proc  1 (commander) line  71 "pan_in" (state -)     [values: 2!cancel,1,1]
132:    proc  1 (commander) line  71 "pan_in" (state 44)
[neighbours[i]!cancel,commander_c,1]
134:    proc  2 (soldier) line 140 "pan_in" (state -)       [values: 2?cancel,1,1]
134:    proc  2 (soldier) line 140 "pan_in" (state 51)
[soldier_c[myid]?cancel,sender,every]
136:    proc  2 (soldier) line 141 "pan_in" (state 52)      [codePositionLEP =
10]
138:    proc  2 (soldier) line 143 "pan_in" (state 53)      [(every)]    <merge 0
now @54>
138:    proc  2 (soldier) line 144 "pan_in" (state 54)      [i = 0]
140:    proc  2 (soldier) line 149 "pan_in" (state 58)      [else]
142:    proc  2 (soldier) line  94 "pan_in" (state 3)       [codePositionLEP = 6]
144:    proc  1 (commander) line  72 "pan_in" (state 45)
[nr_cancel_commander = (nr_cancel_commander+1)]
146:    proc  1 (commander) line  73 "pan_in" (state 46)    [i = (i+1)]
148:    proc  1 (commander) line  60 "pan_in" (state 31)    [i = (i+1)]
spin: trail ends after 148 steps
#processes: 5
148:    proc  4 (soldier) line  95 "pan_in" (state 66)
148:    proc  3 (soldier) line  95 "pan_in" (state 66)
148:    proc  2 (soldier) line  95 "pan_in" (state 66)
148:    proc  1 (commander) line  57 "pan_in" (state 34)
148:    proc  0 (:init:) line 162 "pan_in" (state 5)
5 processes created
```

Figura 3.15: Versão textual de parte do contra-exemplo retornado pelo Spin

SMV

Nesta subseção apresentaremos como processamos as saídas do SMV para retornar os contra-exemplos no nível de abstração de LEP.

O módulo que faz a conversão do Código Intermediário para SMV (*CI2SMV* na figura 3.2) tem uma tarefa mais árdua que o anterior, para Promela. Isto porque a linguagem do SMV é baseada em sistemas de transição, enquanto que Promela é baseada em processos, assim como LEP. Entretanto, a recuperação dos contra-exemplos funciona de forma mais simples, pois o verificador atualiza as variáveis de estado dos processos, assim como nosso marcador (*codePositionLEP*). Assim, para cada estado alcançado, teremos o ponto correspondente na especificação em LEP.

Dada a fórmula no Código Intermediário já apresentada anteriormente,

[] (not(commander[1]!cancel) and not(soldier[1]!cancel) and
not(soldier[2]!cancel) and not(soldier[3]!cancel))

a fórmula temporal equivalente em CTL é:

AG (p[1].state != cancel and p[2].state != cancel and p[3].state != cancel and p[4].state != cancel)

, onde *state* é variável local de um módulo que indica o estado onde este se encontra. Para esta fórmula, obtemos o contra-exemplo apresentado na figura 3.16.

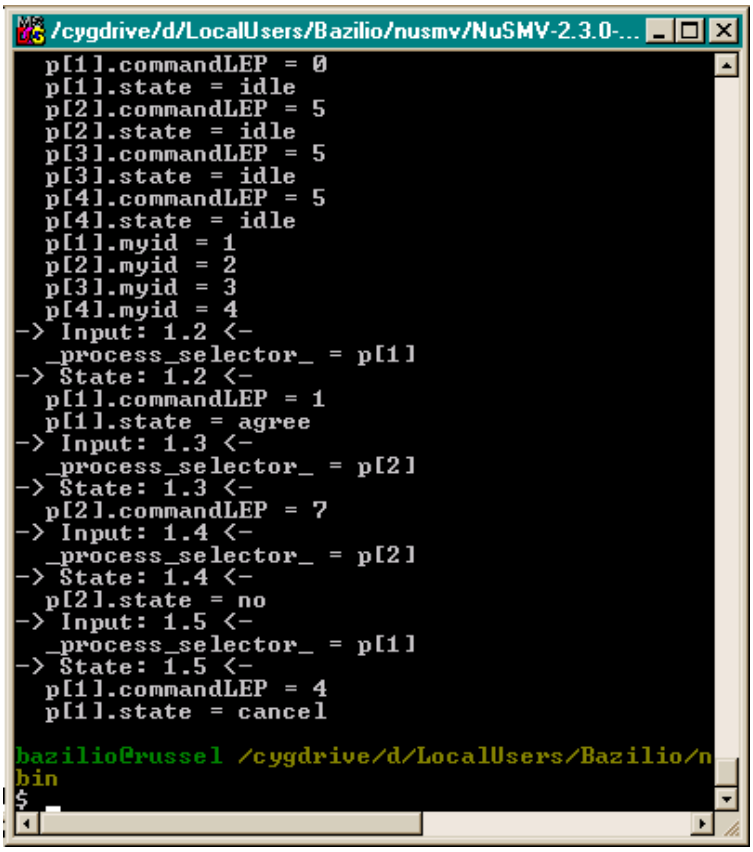

Figura 3.16: Contra-exemplo retornado pelo SMV (*commandLEP* é o mesmo que *codePositionLEP*

Assim, podemos utilizar o método descrito na subseção anterior de maneira análoga para obtermos o contra-exemplo no nível de LEP listado abaixo.

```
commander!agree(everyone) =>
        commander?no =>
                commander!cancel(everyone)
```

Tanto o contra-exemplo apresentado na figura 3.14, quanto o da figura 3.16 não são difíceis de serem interpretados, já que possuem poucos nós. Entretanto, se adicionarmos mais elementos, o número de mensagens trocadas cresce sensivelmente, dificultando bastante a interpretação. Além disso, a recuperação de contra-exemplos no nível de LEP tem a vantagem de termos

um mesmo contra-exemplo para diferentes verificadores de modelos, ou de encontrarmos cenários diferentes que resultem no mesmo contra-exemplo.