DIÓGENES COGO FURLAN

ESPECIFICAÇÃO FORMAL DO SNMPv3 USANDO SEMÂNTICA DE AÇÕES

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Martin A. Musicante

Co-orientador: Prof Elias P. Duarte Jr.

CURITIBA

2000

UFPR
Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Comissão Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Diógenes Cogo Furlan, avaliamos o trabalho intitulado "**Especificação Formal do SNMPv3 Usando Semântica de Ações**", cuja defesa foi realizada no dia 21 de julho de 2000. Após a Avaliação, decidimos pela Aprovação do Candidato.

Curitiba, 21 de julho de 2000.

Prof. Dr. Martin Alejandro Musicante
Presidente - DINF/UFPR

Prof. Dr. Elias Procópio Duarte Jr.
DINF/UFPR

Prof. Dr. Hermano Perrelli de Moura.
UFPE

Profª. Dra. Cristina Duarte Murta
DINF/UFPR

DIÓGENES COGO FURLAN

ESPECIFICAÇÃO FORMAL DO SNMPv3 USANDO SEMÂNTICA DE AÇÕES

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Martin A. Musicante
Co-Orientador: Prof. Elias P. Duarte Jr.

CURITIBA
2000

AGRADECIMENTOS

Ao meu orientador Prof. Martin A. Musicante pelas muitas reuniões, pelos materiais que me ensinou a ler e interpretar, e por ficar surpreso quando eu pouco perguntava e o problema resolvido trazia.

Ao meu co-orientador Prof. Elias P. Duarte Jr. pelas divertidas conversas e pelo apoio e colaboração dados à produção deste trabalho. Pela motivação dada em suas aulas ministradas sobre redes de computadores.

Ao meu mais que amigo Prof. Alexandre Direne pela confiança que sempre depositou em mim, pelo apoio dado quando iniciei a lecionar na faculdade e pelas conversas agradáveis que ajudaram o tempo a passar melhor.

Ao prof. Urban pelas curiosas conversas filosóficas na sala do café.

A todos os professores do Depto. de Informática da UFPR que me apoiaram e confiaram em mim.

À amiga Josiane por toda sua ajuda e dedicação ao estudo de Semântica de Ações.

Ao amigo Aldri por toda sua ajuda que aumentou meu conhecimento sobre Latex, sobre Linux e por indicar um bom maracujá para o dia da defesa.

Aos amigos Ionildo, Denis, Denio, Mozart, Rudá, Simone, Inali, Luciane e Patrícia pela confiança e dedicação durante este período de nossas vidas.

À minha eterna namorada, atual noiva e futura mulher, Luciana, por todo seu amor, carinho e dedicação neste longo período de produção desta dissertação. Pelos capítulos que me ajudou a completar e pela força que me trouxe nos momentos mais difíceis.

Aos meus pais por ter nascido.

A Deus por ter me criado e permitido que tudo isto acontecesse.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Semântica de Ações é um formalismo utilizado para prover descrições legíveis de linguagens de programação, devido sua estrutura modular e sua notação formal baseada em termos da língua inglesa.

Simple Network Management Protocol (SNMP) é o sistema de gerência de redes padrão da Internet, estando especificada sua terceira versão (SNMPv3).

Este trabalho consiste na leitura e interpretação dos documentos que definem informalmente o SNMPv3, e na sua definição formal em semântica de ações. Esta especificação tem a finalidade de melhorar a comunicação entre seres humanos (definindo de forma não ambígua a semântica da entidade e dos objetos de gerência de redes), possibilitar a verificação de implementações já existentes e possibilitar a geração de implementações automáticas de sistemas de gerência de redes.

A especificação formal do SNMP usando semântica de ações consiste em especificar as operações do protocolo, os componentes da entidade SNMP (incluindo o Despachante e as aplicações padrão) e fornecer suporte para futuras especificações formais de informações de gerência.

A especificação das operações SNMP contém as ações que correspondem aos procedimentos seguidos quando se gera e envia uma operação de gerência ou quando se processa sua resposta. A especificação do Despachante SNMP é formada por ações que correspondem aos procedimentos seguidos no envio de mensagens de requisição e de resposta para outras entidades, e na recepção de mensagens de requisição e de resposta vindas de outras entidades. A especificação das aplicações padrão do SNMP contém as ações representando os procedimentos seguidos para gerar e enviar ou receber e processar operações SNMP. A especificação de uma entidade SNMP é formada por ações correspondentes aos módulos de uma entidade SNMP, para compor entidades atuando como gerente SNMP (responsáveis por controlar e monitorar objetos de gerência) ou como agente SNMP (responsáveis por manter objetos de gerência).

Neste trabalho é apresentada, também, uma análise crítica da especificação informal do SNMP, e a identificação de um conjunto de características problemáticas da mesma.

ABSTRACT

Action semantics is a framework to allow useful semantic descriptions of realistic programming languages.

The Simple Network Management Protocol version 3 (SNMPv3) is the Internet standard management architecture.

This work presents a formal description of SNMPv3 using action semantics. The main goals of this description are to contribute to the accurate understanding of SNMP entities, to allow the verification of existent SNMP implementations, as well as to set the basis for the automatic generation of SNMP implementations.

The formal specification of SNMP using action semantics consists in the specification of the protocol operations, the entity modules (including the dispatcher and the standard applications) and sets the basis for future specifications of management information.

The formal specification of SNMP operations contains actions to generate and send a requisition and actions to process its response. The formal specification of the SNMP dispatcher is composed by actions to send requisitions and response messages to other entities, as well as actions to receive the messages coming from other entities. The formal specification of SNMP standard applications consists in actions to generate and send or actions to receive and process SNMP operations. The formal specification of SNMP entity is composed by actions that represents the several modules of an entity, to generate managers (responsible to control and monitoring management objects) or agents (responsible to store management objects).

Other result of this work is a critical analysis of the SNMP informal specification, concluding in the identification of a number problematic items in it.

CAPÍTULO 1

Introdução

Pelo fato das redes de computadores terem se tornado muito populares e ao mesmo tempo maiores e mais complexas, a necessidade de *sistemas integrados de gerência de redes* tornou-se crítica. O propósito destes sistemas é auxiliar gerentes humanos no trabalho do monitoramento e controle de redes de computadores. *Simple Network Management Protocol* (SNMP) é o sistema de gerência de redes padrão da Internet, estando especificada sua terceira versão (SNMPv3) [HPW99, CHPW99, LMS99]. Há atualmente um grande número de sistemas disponíveis baseados no SNMP, tanto comerciais quanto de domínio público.

Atualmente, a semântica dos componentes do SNMP é dada informalmente em uma série de textos em inglês explicando o significado de cada um deles [CMRW96b, CMPS99, HPW99, CHPW99, LMS99, BW99, WPM99, MPS99b, MPS99c, MPS99a]. A natureza informal destas descrições pode levar a interpretações incorretas e abrir a possibilidade de implementações inconsistentes.

Descrições formais podem ser usadas como base para um melhor entendimento do conteúdo a ser definido, para provar propriedades, para comunicar idéias de maneira precisa a quem desenvolve uma aplicação, e para verificar e gerar implementações automaticamente.

Semântica de Ações é um formalismo introduzido em [Mos92, Wat91] para prover descrições legíveis de linguagens de programação. Descrições em semântica de ações mapeiam sintaxe abstrata em entidades *ad hoc* chamadas *ações*, que podem ser executadas. Descrições em semântica de ações são modulares e facilmente extensíveis ou modificáveis. Ações permitem separar o fluxo de controle dos diversos fluxos de dados durante a execução, provendo uma forma natural para descrever computações.

Outros formalismos já foram propostos para especificar linguagens de programação, como *Semântica Denotacional* [Sch86], *Semântica Operacional* [Win93], *Semântica Algébrica* [Wat91], entre outros. Semântica de ações foi desenvolvida a partir da semântica denotacional, e trata aspectos pragmáticos desejáveis, que são problemáticos neste formalismo, como dificuldade de

leitura e ausência de modularidade. Por fim, semântica de ações é um formalismo capaz de expressar a maioria dos conceitos usados na computação, pois foi produzido com este objetivo.

1.1 Objetivos

Este trabalho consiste na leitura e interpretação dos documentos que definem informalmente o SNMPv3, e na sua definição formal em semântica de ações, incluindo:

- as operações do protocolo;
- os componentes da entidade SNMP, incluindo o Despachante e as aplicações padrão;
- suporte para futuras especificações formais de informações de gerência.

Este trabalho tem a finalidade de:

- melhorar a comunicação entre seres humanos definindo de forma não ambígua a semântica dos objetos e das entidades SNMP;
- permitir a verificação de implementações já existentes;
- possibilitar a geração de implementações automaticamente.

1.2 Organização da Dissertação

Esta dissertação está organizada em nove capítulos. O Capítulo 2 apresenta uma introdução à semântica de ações, enfatizando as ações que serão mais utilizadas no decorrer do trabalho. O Capítulo 3 revisa a arquitetura padrão do SNMPv3, apresentando a forma pela qual as informações de gerência são manipuladas, quais são as operações do protocolo e quais são os tipos de entidades e sua composição. O Capítulo 4 traz uma descrição concisa dos trabalhos correlatos a esta dissertação. O resto da dissertação trata da especificação do SNMPv3 usando semântica de ações: o Capítulo 5 define a especificação das operações SNMP; o Capítulo 6 descreve a especificação do Despachante SNMP; o Capítulo 7 mostra a especificação das aplicações padrão do SNMP; o Capítulo 8 trata sobre a especificação de uma entidade SNMP. O Capítulo 9 contém as conclusões.

CAPÍTULO 2

Semântica de Ações

Semântica de Ações [Mos92] é um formalismo utilizado para prover descrições semânticas de linguagens de programação de uso real, além de outras aplicações computacionais.

Muitos formalismos têm sido propostos para especificar linguagens de programação; tipicamente, cada um deles enfatiza e facilita diferentes aspectos e aplicações destas linguagens. Semântica de ações foi desenvolvida a partir da *Semântica Denotacional* [Sch86], e trata aspectos pragmáticos desejáveis que aquele formalismo ignora, como facilidade de leitura, modularidade, capacidade de tratar abstrações, capacidade de comparar diversas especificações produzidas em um mesmo formalismo e facilidade de provar propriedades algébricas que facilitem transformações.

Descrições semânticas são essencialmente compostas pelo mapeamento de entidades sintáticas em entidades semânticas. Semântica de ações usa uma notação especial para descrever entidades semânticas, chamada de *notação de ações*. Esta notação formal é utilizada da mesma forma que a notação-λ é utilizada na semântica denotacional.

A Seção 2.1 apresenta este formalismo utilizado para descrever ações e dados em semântica de ações. A Seção 2.2 descreve como as principais características da computação são abordadas, mostrando as ações definidas pelo formalismo, além de exemplos de utilização destas ações.

2.1 Notação de Ações

Semântica de ações descreve a semântica de linguagens e aplicações usando uma metalinguagem formal denominada *Notação de Ações*. Os símbolos usados na notação de ações são intencionalmente prolixos, sendo utilizados termos da língua inglesa - mas completamente formais, pois funcionam como funções - para expressar a maioria dos conceitos presentes na computação (facilidade de leitura).

A notação de ações é definida em [Mos92]. Esta notação pode ser facilmente estendida através da inclusão de novos tipos de dados ou através da criação de abreviaturas para entidades já

especificadas (modularidade).

Há três tipos de entidades na notação de ações: *ações*, *dados* e *produtores*. Ações são essencialmente dinâmicas: entidades computacionais representando controle e processamento de informação. Dados, em contraste, são essencialmente estáticos: entidades matemáticas representando peças de informação. Um produtor representa um dado não avaliado, cujo valor depende da informação correntemente disponível como entrada à ação onde o produtor está inserido.

Descrições em Semântica de Ações (ASD) consistem em especificações divididas nos seguintes módulos:

- *Entidades sintáticas*, que incluem a sintaxe concreta (*parser*) e abstrata (estrutura) dos programas de uma linguagem. Muitas vezes, apenas a sintaxe abstrata é definida. Gramáticas livres de contexto são utilizadas para a sua especificação.

- *Funções semânticas*, que são usadas para mapear entidades sintáticas em entidades semânticas. Estas funções são composicionais, mapeando frases complexas em função dos mapeamentos definidos para as suas frases componentes. Equações semânticas são utilizadas para a sua especificação.

- *Entidades semânticas*, parte da especificação que define dados e ações auxiliares usados pelas funções semânticas.

Em semântica de ações definem-se diversos *tipos de informações*, que são as diferentes interpretações que os dados recebem dependendo do contexto em que eles se encontrem, incluindo a forma pela qual os dados são manipulados e a forma pela qual eles tendem a se propagar. Sua classificação é a seguinte:

- *informação transitória*: representa os dados que são passados de uma ação para outra de forma funcional. Este tipo de informação corresponde à avaliação de expressões em linguagens de programação.

- *informação com escopo*: representa associações de identificadores a dados, com a finalidade de vincular um nome àquele dado; tal associação é denominada *binding*. Este tipo de informação corresponde ao ambiente de definição de identificadores utilizado nas linguagens de programação.

- *informação estável*: representa dados armazenados em células de memória. Este tipo de informação corresponde aos valores atribuídos às variáveis em linguagens de programação.

- *informação permanente*: representa dados que são trocados entre diversas ações num ambiente de execução distribuído. Este tipo de informação corresponde às mensagens trocadas entre processos ou entre máquinas distintas. É definido um local de armazenamento (*buffer*) para a sua recepção.

2.1.1 Ações

Ações são entidades que podem ser *executadas* e seu desempenho corresponde a possíveis comportamentos da computação, que pode tanto:

- *completar*, correspondendo à terminação normal (estado *completing*);
- *escapar*, correspondendo à terminação excepcional (estado *escaping*);
- *falhar*, correspondendo à terminação sem sucesso ou ao abandono da execução; a maioria das ações que usam informação falham quando a informação necessária não está disponível durante a sua execução (estado *failing*);
- *divergir*, correspondendo à não-terminação (estado *diverging*).

Uma ação interage com as outras recebendo informações antes de sua execução e fornecendo informações após. Como visto anteriormente os tipos de informação processados pelas ações são: transitória, de escopo, estável e permanente.

Informação transitória é disponibilizada a uma ação para uso imediato. Informação de escopo, em contraste, pode ser propagada durante a execução de uma ação e até ser omitida temporariamente quando uma nova associação a um identificador já existente for gerada. Informação estável pode ser modificada, mas não omitida, e ela persiste até ser explicitamente destruída. Informação permanente não pode ser modificada, mas somente ampliada.

Após a execução de uma ação, a informação transitória produzida é disponibilizada quando a ação completa ou escapa, e nunca é gerada quando a ação falha ou diverge. Quanto à informação com escopo, ela sempre é produzida quando a ação completa, e nunca é produzida quando a ação escapa, falha ou diverge.

Se uma ação não pode ser decomposta em ações menores na semântica de ações, ela é dita ser uma *ação primitiva*. *Combinadores de ações* possibilitam canalizar o fluxo de informações de maneiras diferentes entre as ações. *Ações compostas* são as ações geradas pela ligação de uma ou mais ações através de combinadores de ações; estas ações são então consideradas como *subações* da ação composta.

A execução de uma ação consiste de uma sequência de passos atômicos executados por um único agente[1] (definido na página 14). A execução de uma ação primitiva consiste de um único passo. A execução de ações compostas pela combinação de outras ações pode ser: *sequencial*, quando todos os passos de uma ação são executados por completo antes que a execução da segunda ação inicie; *intercalado*, quando os passos de ambas as ações são executados em uma ordem arbitrária; *exclusivo*, quando somente uma das ações é executada.

2.1.2 Dados

A *notação de dados* é usada para descrever os dados processados pelas ações. A notação de dados incluída na notação de ações padrão provê uma coleção de tipos de dados definidos algebricamente, incluindo números, valores-verdade, caracteres, *strings*, listas, conjuntos, tuplas, mapeamentos, etc.; também são incluídos identificadores, células e agentes, que são utilizados para acessar outros dados, e algumas entidades compostas com componentes de dados, tais como mensagens e contratos. Ações, por si só, não são dados, mas podem ser incorporadas em *abstrações*, que são dados, e subsequentemente extraídas de volta em ações. Tipos de dados adicionais podem ser especificados conforme a necessidade.

Todos os dados da semântica de ações são especificados algebricamente usando *álgebras unificadas* [Mos89], um modelo algébrico não convencional onde a diferença entre *sorte*[2] e elemento é eliminada. Elementos individuais não são distinguidos de conjuntos contendo apenas aquele elemento.

2.1.3 Produtores

Produtores são entidades que podem ser *avaliadas* para gerar dados. Um produtor tem um desempenho com relação ao dado produzido, que pode:

- *resultar em um item de dados*, quando o dado é gerado normalmente;
- *resultar em* nothing, quando o dado não é gerado porque uma das pré-condições necessárias para que isto ocorresse não foi satisfeita, ou quando o dado gerado não possui o tipo esperado.

[1] Apesar do termo *agente* aparecer tanto em SNMP quanto em Semântica de Ações, eles têm significados diferentes: em SNMP, um agente é a entidade que contém informações de gerência; em Semântica de Ações, um agente é uma entidade autônoma que executa uma ação.

[2] Conjunto de indivíduos que possuem propriedades em comum.

Além das próprias constantes e operadores de dados poderem ser classificados como produtores, existem produtores específicos para transformar os diversos tipos de informação processada (transitória, com escopo, estável, permanente) em dados. Por exemplo, o produtor current storage resulta no mapeamento que representa a memória no momento em que o produtor for chamado.

2.2 Facetas

O comportamento das ações é dividido em facetas, conforme o tipo de informação por elas processado:básica, funcional, declarativa, imperativa, reflexiva e comunicativa.

Cada faceta provê algumas ações primitivas e combinadores para formar ações complexas, correspondendo aos principais conceitos fundamentais da computação. Além disto, provê um mecanismo para manipular dados e para gerar dados partindo de um tipo específico de informação.

2.2.1 Faceta Básica

Faceta básica é aquela cujos componentes estão relacionados apenas com o *fluxo de controle* na execução de diversas ações.

As ações descritas na faceta básica são:

- complete: é uma ação indivisível que (simplesmente) termina normalmente. Ela corresponde a uma ação nula.
- escape: é uma ação indivisível que termina de forma excepcional.
- fail: é uma ação indivisível que termina com falha.
- diverge: é uma ação que nunca termina. De fato, diverge é uma abreviatura para unfolding unfold.
- commit: ação similar com complete, porém com o efeito colateral de sinalizar uma alteração nula na informação estável ou permanente processada pela ação.
- unfold: é uma ação auxiliar usada juntamente com o combinador unfolding para desviar o fluxo de controle de volta para o início deste combinador.

Os combinadores de ações descritos na faceta básica são:

- A_1 and then A_2: corresponde à execução sequencial das ações A_1 e A_2, executando A_2 somente quando terminar a execução de A_1. Seu elemento neutro é complete.

- A_1 and A_2: corresponde à execução intercalada das ações A_1 e A_2. Seu elemento neutro é complete.

- A_1 trap A_2: é um combinador que permite que uma ação auxiliar A_2 seja executada quando a ação A_1 terminar excepcionalmente. Seu elemento neutro é escape.

- A_1 or A_2: introduz a idéia de escolha não-determinística e limitada de ações. Uma das ações A_1 ou A_2 é escolhida ao acaso e somente se esta ação falhar a outra será executada. No caso de uma ação commit ter sido executada, a ação alternativa fica descartada e o combinador or retorna falha no caso de falha na execução da primeira ação. Seu elemento neutro é fail.

- unfolding A: a ação A será executada normalmente até o momento de executar a ação unfold, quando esta será substituída pela própria ação A.

- indivisibly A: a ação A é desempenhada como se fosse composta apenas por um único passo.

Exemplos:

- A seguinte ação sempre falha, pois a ação fail é sempre a última a ser executada.

```
| complete
and then
| fail
```

- A seguinte ação pode falhar em alguns casos, mas em outros ela escapa, pois as duas subações são executadas em uma ordem não determinística.

```
| escape
and
| fail
```

- A seguinte ação corresponde a um *loop* sem condição explícita de parada executando a ação A.

```
unfolding
| | A
| and then unfold
```

- A seguinte ação termina normalmente quando a ação A sinalizar uma exceção.

```
| A
trap complete
```

2.2.2 Faceta Funcional

Faceta funcional é aquela cujos componentes estão relacionados com o *fluxo de informação transitória*, que são tuplas de dados trocadas entre as ações. Cada campo de uma tupla é identificado por um rótulo numérico que simboliza a sua posição na tupla.

Os dados avaliados como informação transitória são definidos formalmente como indivíduos do sorte data, que são tuplas contendo zero ou mais indivíduos do sorte datum. O sorte datum abrange os sortes numéricos, *truth-value*, caracter, *string*, entre outros [Mos92, Capítulo 5].

As ações descritas na faceta funcional são:

- give Y: resulta em informação transitória com os dados fornecidos pelo produtor Y.
- escape with Y: escapa retornando como informação transitória os dados fornecidos pelo produtor Y; esta informação pode ser usada para identificar o motivo do escape.
- check Y: testa se o dado fornecido pelo produtor Y é igual ao sorte true, ou seja, se é verdadeiro, terminando normalmente; caso contrário, falha.
- choose Y: retorna um único elemento dos dados gerados pelo produtor Y.
- regive: propaga toda informação transitória recebida. É uma abreviatura para give the given data.

Os produtores descritos na faceta funcional são:

- given R: resulta em dados a partir de toda informação transitória recebida para ser avaliada, desde que esta informação seja do sorte R.
- given R # N: resulta em dados a partir do N-ésimo componente da informação transitória recebida para ser avaliada, desde que esta informação seja do sorte R.
- it: produtor para uma peça de informação, sem importar o seu tipo. É uma abreviatura para the given datum.
- then: produtor para toda informação fornecida, sem importar o seu tipo. É uma abreviatura para the given data.

O combinador de ações descrito na faceta funcional é:

- A_1 then A_2: corresponde à composição funcional da informação transitória, permitindo que somente as informações transitórias geradas pela subação A_1 sejam propagadas para a subação A_2. Seu elemento neutro é regive.

Exemplos:

- A seguinte ação completa quando a informação transitória recebida for do tipo especificada pelo produtor the given. Caso contrário, ela falha.

```
give the given integer
```

- A seguinte ação representa um condicional que executa a ação A caso o teste X devolva verdadeiro e executa a ação B caso o teste X devolva falso.

```
| | check X
| and then
| | A
or
| | check not X
| and then
| | B
```

- A seguinte ação corresponde a uma iteração com número definido de ciclos (N), executando a ação A.

```
give N and then
unfolding
| | check (the given integer is 0)
| or
| | | check (the given integer is greater than 0)
| | and then
| | | A
| | and then
| | | give difference(the given integer, 1) then unfold
```

2.2.3 Faceta Declarativa

Faceta declarativa é aquela cujos componentes estão relacionados com o *fluxo de informação com escopo*, que é aquela proveniente da ligação de identificadores a dados identificáveis. Esta associação é denominada de *binding*.

Os dados avaliados como informação com escopo são definidos formalmente como indivíduos do sorte bindings, que são as associações de identificadores a valores identificáveis. Os identificadores são definidos como elementos do sorte token. Os valores que podem ser associados aos *tokens* são pertencentes ao sorte bindable (definido em cada descrição em semântica de ações na seção de entidades semânticas). Um outro valor que pode ser associado a um *token* é o unknown, que indica um *binding* desconhecido.

As ações descritas na faceta declarativa são:

- bind K to Y: produz como informação com escopo o *binding* do identificador K para o dado avaliado pelo produtor Y.

- produce Y: produz *bindings* para o mapeamento de identificadores a valores do sorte bindable fornecido pelo produtor Y.
- unbind K: desfaz o *binding* relativo ao identificador K. É uma abreviatura para bind K to unknown.
- rebind: propaga todos os *bindings* recebidos para o próximo escopo. É uma abreviatura para produce the current bindings.

Os produtores descritos na faceta declarativa são:

- current bindings: resulta no mapeamento correspondente à informação com escopo do ambiente corrente.
- the R bound to K: resulta como dado a informação associada ao identificador K, desde que este *binding* esteja contido na informação com escopo recebida e desde que o valor associado ao identificador K seja do sorte R.
- Y_1 receiving Y_2: retorna os dados produzidos pelo produtor Y_1, tendo este recebido os *bindings* produzidos pelo produtor Y_2.

Os combinadores de ações descritos na faceta declarativa são:

- A_1 moreover A_2: a informação com escopo resultante deste combinador será composta pela união das informações com escopo produzidas pelas subações A_1 e A_2. Caso alguma associação seja gerada em ambas subações, então valerá a produzida pela subação A_2. As ações A_1 e A_2 são executadas intercaladamente. Seu elemento neutro é complete.
- A_1 hence A_2: corresponde à composição funcional da informação com escopo, permitindo que somente as informações com escopo geradas pela subação A_1 sejam propagadas para a subação A_2. As ações A_1 e A_2 são executadas sequencialmente. Seu elemento neutro é rebind.
- A_1 before A_2: a informação com escopo recebida pela ação A_1 é a recebida pelo combinador; já a informação com escopo recebida pela ação A_2 é formada pela junção da informação com escopo gerada pela ação A_1 com a recebida pelo combinador. A informação com escopo gerada pelo combinador será a junção da informação com escopo gerada pelas duas subações. As ações A_1 e A_2 são executadas sequencialmente. Seu elemento neutro é complete.
- furthermorer A: é uma abreviatura para rebind moreover A.

Exemplos:

- A seguinte ação cria a associação do valor 1 ao identificador *"id"*.

```
bind "id" to 1
```

- A seguinte ação retorna o valor associado ao identificador *"id"*.

```
give the integer bound to "id"
```

2.2.4 Faceta Imperativa

Faceta imperativa é aquela cujos componentes estão relacionados com o *fluxo de informação estável*.

Os dados avaliados como informação estável são definidos formalmente como indivíduos do sorte storage, que são mapeamentos de células para valores armazenáveis. A memória é formado por um conjunto (potencialmente infinito) de células, que são definidas como elementos do sorte cell. Em um determinado momento, cada célula de memória pode estar alocada ou não, e se estiver alocada, poderá armazenar dados. Os valores que podem ser armazenados em células são pertencentes ao sorte storable (definido nas entidades semânticas de cada ASD). Um outro valor que pode ser vinculado a uma célula é o uninitialized, que indica que nenhum dado está armazenado naquela célula.

As ações descritas na faceta imperativa são:

- store Y_1 in Y_2: produz como informação estável o armazenamento do valor fornecido pelo produtor Y_1 na célula fornecida pelo produtor Y_2, desde que esta célula faça parte da informação estável já reservada.

- unstore Y: remove os dados armazenados na célula fornecida pelo produtor Y, desde que esta célula faça parte da informação estável já reservada.

- reserve Y: reserva a célula fornecida pelo produtor Y para uso, desde que esta célula não faça parte da informação estável já reservada.

- unreserve Y: desaloca a célula fornecida pelo produtor Y, desde que esta célula faça parte da informação estável já reservada.

Os produtores descritos na faceta imperativa são:

- current storage: resulta no mapeamento correspondente à informação estável recebida pela ação.

- the R stored in Y: resulta em um dado com a informação armazenada na célula de memória retornada como dado pelo produtor Y, desde que o produtor Y retorne uma célula que faça parte da informação estável já reservada e desde que o valor armazenado nesta célula seja do sorte R.

A seguinte ação é composta por outras ações descritas anteriormente e têm a finalidade de simplificar a escrita de novas ações:

- allocate a cell: escolhe uma célula dentre as células não alocadas, faz sua reserva e a devolve como informação transitória. É uma abreviatura para:

```
allocate a cell =
    indivisibly
    | | choose a cell [not in mapped-set of the current storage]
    then
    | | reserve the given cell and give it
```

Exemplos:

- A seguinte ação aloca uma nova célula de memória não ocupada, e associa esta célula ao identificador *"id"*.

```
| allocate a cell
then
| bind "id" to the given cell
```

- A seguinte ação armazena o valor 10 na célula associada ao identificador *"id"*.

```
store 10 in the cell bound to "id"
```

2.2.5 Faceta Reflexiva

Faceta reflexiva é aquela cujos componentes estão relacionados com o *encapsulamento de ações como dados.*

A faceta reflexiva define o sorte abstraction para representar as abstrações.

A ação descrita na faceta reflexiva é:

- enact Y: executa a ação incorporada na abstração fornecida pelo produtor Y.

Os produtores descritos na faceta reflexiva são:

- abstraction of A: resulta em um dado a partir do encapsulamento da ação A.
- application A to Y: resulta em um dado a partir do encapsulamento da ação A juntamente com a informação transitória proveniente como resultado fornecido pelo produtor Y.
- closure A: resulta em um dado a partir do encapsulamento da ação A juntamente com a informação com escopo recebida.

2.2.6 Faceta Comunicativa

Faceta comunicativa é aquela cujos componentes estão relacionados com o *processamento de informação em sistemas distribuídos de agentes.*

Agente é a identidade de entidades responsáveis pela execução das ações. Agentes se comunicam de forma assícrona através da passagem de mensagens definidas pelo sorte message. Uma mensagem contém, entre outras coisas, o remetente para o qual ela é destinada e os dados que serão transferidos. Os dados que podem ser trocados em mensagens são do sorte sendable, que inclui abstrações e agentes.

Ações podem ser relacionadas a agentes, para serem executadas. Agentes podem ser criados dinamicamente através da oferta de contratos definidos pelo sorte contract. Um contrato contém o agente a ser contratado e a ação a ser executada por este agente. Cada agente possui um *buffer*, onde as mensagens são colocadas.

As ações descritas na faceta comunicativa são:

- send Y: transmite a mensagem fornecida pelo produtor Y para o agente especificado na mensagem.
- remove Y: remove a mensagem fornecida pelo produtor Y do *buffer* do agente que está executando a ação.
- offer Y: realiza o contrato fornecido pelo produtor Y.

Os produtores descritos na faceta comunicativa são:

- current buffer: resulta em dados a partir do buffer do agente que está executando a ação.
- performing-agent: resulta como dado a identidade do agente que está executando a ação.
- contracting-agent: resulta como dado a identidade do agente que contratou o agente que está executando a ação.

O combinador descrito na faceta comunicativa é:

- patiently A: representa espera ocupada enquanto a ação A falha, ou seja, a ação A é executada enquanto seu estado de terminação for de falha.

As seguintes ações são compostas por outras ações descritas anteriormente e têm a finalidade de simplificar a escrita de novas ações:

- receive Y: espera pela mensagem dada pelo produtor Y até que seja recebida; então a mensagem é removida do *buffer* e retornada como informação transitória. É a abreviatura para:

```
receive Y:yielder =
    patiently
    | | choose a Y [in set of items of the current buffer]
    | then
    | | remove the given message and give it
```

- subordinate Y: cria um novo agente para executar uma ação dada. É a abreviatura para:

```
subordinate Y:yielder =
    | offer a contract[to Y][containing abstraction of subordinate-action]
    and then
    | | receive a message[form Y][containing an agent]
    | then
    | | give the contents of the given message
```

onde subordinate-action é a primeira ação que será executada pelo novo agente. Ela envia a identificação do novo agente para o agente que o criou e espera deste uma ação a ser executada a seguir.

```
subordinate-action =
    | send a message[to the contracting-agent][containing the performing-agent]
    then
    | receive a message[from the contracting-agent][containing an abstraction]
    then
    | enact the contents of the given message
```

Exemplos:

- A seguinte ação envia de forma assíncrona a tupla armazenada na célula associada ao identificador *"D"* numa mensagem, para o agente associado ao identificador *"I"*.

```
send-assynchronous D:token to I:token =
    | give the tuple stored in the cell bound to D
    then
    | send a message[to the agent bound to I][containing the given value]
```

- A seguinte ação recebe de forma assíncrona uma tupla numa mensagem vinda do agente associado ao identificador *"I"*. Esta tupla é então armazenada na célula associada ao identificador *"D"*.

```
receive-assynchronous D:token from I:token =
    | receive a message[from the agent bound to I][containing a tuple]
    then
    | store the contents of the given message in the cell bound to D]
```

- A seguinte ação envia de forma síncrona a tupla armazenada na célula associada ao identificador *"D"* numa mensagem, para o agente associado ao identificador *"I"*.

```
send-synchronous D:token to I:token =
    | give the tuple stored in the cell bound to D
    then
    | | send a message[to the agent bound to I][containing the given value]
    | and then
    | | receive a message[from the agent bound to I][containing ACK]
```

- A seguinte ação recebe de forma síncrona uma tupla numa mensagem vinda do agente associado ao identificador *"I"*. Esta tupla é então armazenada na célula associada ao identificador *"D"*.

```
receive-synchronous D:token from I:token =
    | receive a message[from the agent bound to I][containing a tuple]
    then
    | | send a message[to the agent bound to I][containing ACK]
    | and
    | | store the contents of the given message in the cell bound to D]
```

2.2.7 Comportamento dos Combinadores

Cada combinador da notação de ações lida de forma diferente com os diversos tipos de informações tratadas pela semântica de ações.

A Figura 2.1 apresenta o fluxo de controle para alguns dos combinadores da notação de ações. As setas representam o fluxo de controle dentro do combinador. Em alguns casos, é indicado em quais estados de terminação o controle é passado para uma determinada ação. Os casos sem indicação ocorrem nos demais estados de terminação.

A Figura 2.2 apresenta o fluxo de informação transitória para alguns dos combinadores da notação de ações. As setas representam o fluxo da informação transitória dentro do combinador, indicando a propagação da informação transitória recebida pelo combinador para suas subações e a propagação da informação transitória dada pelas subações. O símbolo $\oplus$ indica o *merge* das informações recebidas; a ação falha em caso de conflito.

A Figura 2.3 apresenta o fluxo de informação com escopo para alguns dos combinadores da notação de ações. As setas representam o fluxo da informação com escopo dentro do combinador, indicando a propagação da informação com escopo recebida pelo combinador para suas subações e a propagação da informação com escopo dada pelas subações. O símbolo $\oplus$ possui o mesmo comportamento explicado na Figura 2.2. O símbolo $\oslash$ indica o *overlay* das informações recebidas, ou seja, os *bindings* criados na ação cuja seta, indicando seu fluxo de saída, atravessa o círculo tem prioridade sobre os da outra ação, caso sejam redefinidos.

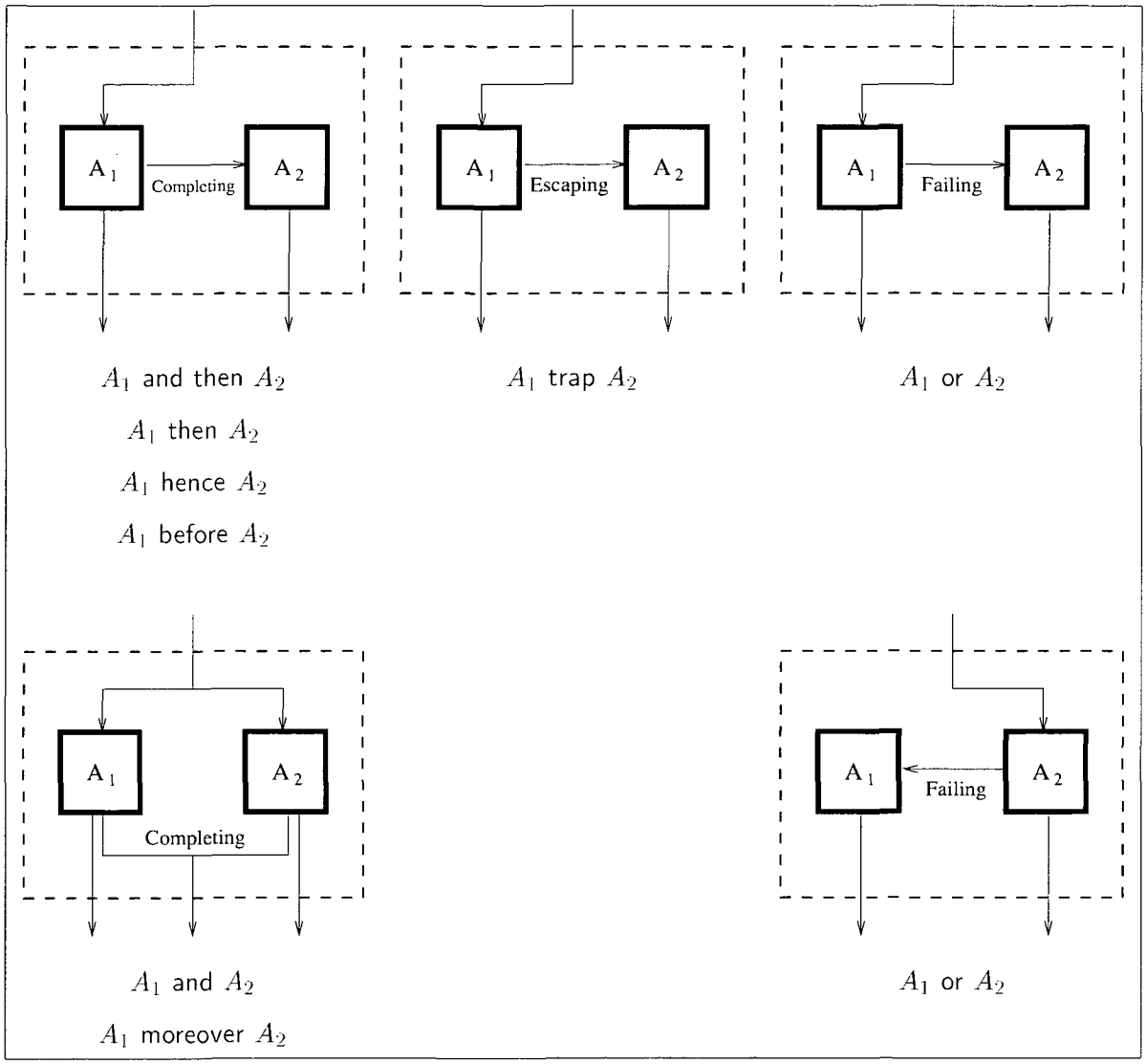

Figura 2.1: Fluxo de controle de alguns combinadores.

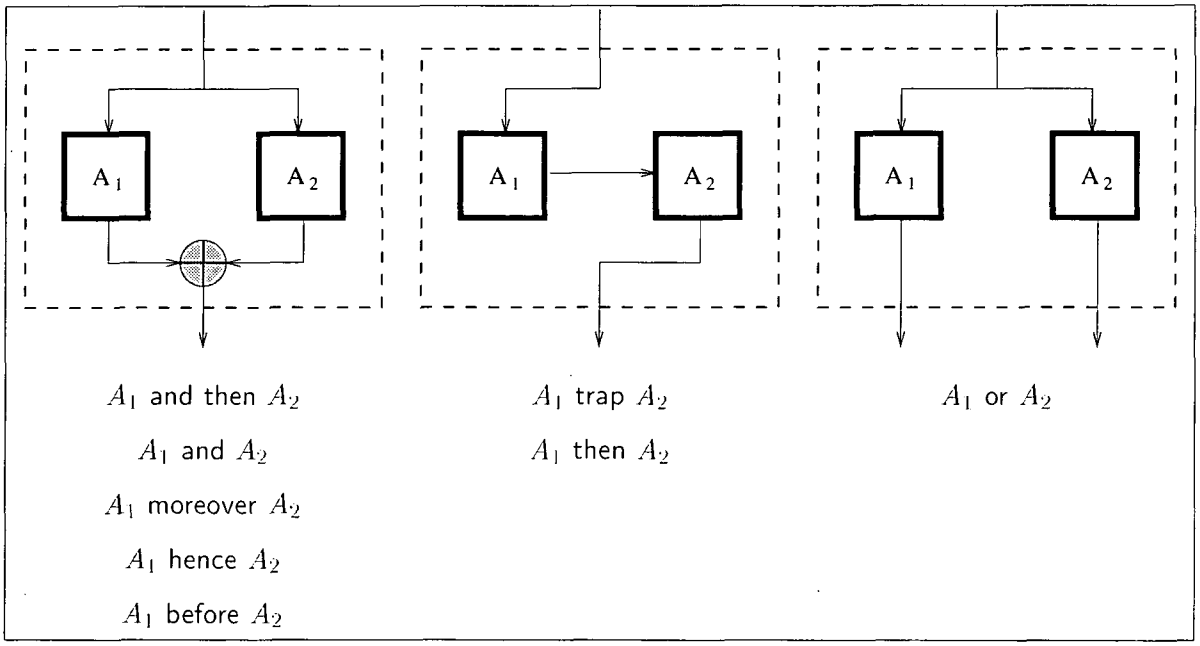

Figura 2.2: Fluxo de informação transitória de alguns combinadores.

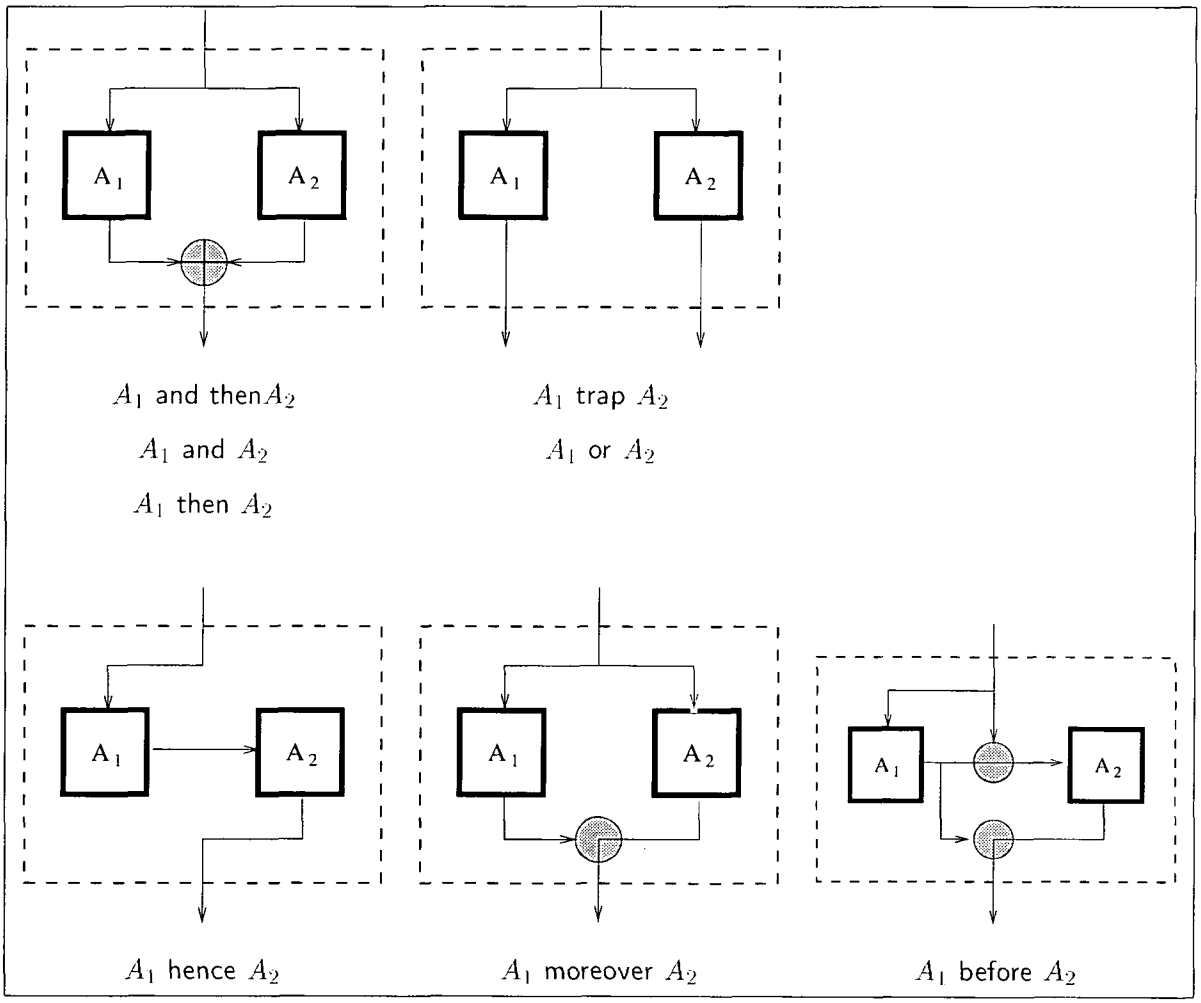

Figura 2.3: Fluxo de informação com escopo de alguns combinadores.

2.3 Considerações Finais

Este capítulo apresentou a essência do formalismo de semântica de ações, que será usado no restante deste trabalho.

Para uma noção mais detalhada de semântica de ações, o leitor pode referir-se a [Mos92]; para uma introdução, veja também [Wat91]. Para uma comparação entre semântica de ações e outros formalismos veja [Mos92, Lee89, Mus96].

Para ter acesso a trabalhos gerando implementações a partir de ações, veja [BMW92], que define o sistema Actress, e [Men98], que define o sistema Abaco.

CAPÍTULO 3

Gerência de Redes Baseada no SNMP

Este capítulo expõe os conceitos e definições relacionadas com a terceira versão do *Simple Network Management Protocol* (SNMPv3). A Seção 3.1 trata sobre gerência de redes de uma forma genérica, expondo as razões e consequências da utilização de mecanismos para gerência de redes. A Seção 3.2 traz a idéia de um sistema de gerência de redes baseado no SNMP. Informação de gerência é abordada na Seção 3.3, as operações de gerência na Seção 3.4 e, por fim, a arquitetura utilizada pelo SNMP, descrevendo seus componentes e funcionalidades, é mostrada na Seção 3.5.

3.1 Gerência de Redes de Computadores

Uma *rede de computadores* [LC96] é uma coleção de dispositivos e circuitos para a transferência de dados entre computadores. Isto permite que usuários em diferentes localidades possam compartilhar recursos de um computador remoto.

A *gerência de redes de computadores* [Sta98b, Ros94] consiste em atividades de controle e monitoramento dos dispositivos de *hardware* ou de *software* que constituem uma rede de computadores.

Estes dispositivos gerenciados podem ser computadores, roteadores, pontes, *hubs*, *modems*, terminais, impressoras ou outro dispositivo que possa disponibilizar alguma informação à rede, assim como aplicações.

Para melhor definir o escopo da gerência de redes, a *International Organization for Standardization* (ISO) dividiu sua funcionalidade em cinco áreas [LC96, Sta98b]: gerência de falhas, gerência de desempenho, gerência de configuração, gerência de segurança e gerência de contas, descritas a seguir:

- *Gerência de falhas* é o processo de localizar e corrigir problemas na rede, ou *falhas*. Ela consiste em identificar a ocorrência de falhas, isolar sua causa e corrigí-la (quando possível). Um exemplo de falha na rede é um *link* que está inativo.

- *Gerência de desempenho* envolve medições do desempenho do *hardware*, *software* e meios de transmissão de uma rede. Usando informações da gerência de desempenho, o gerente da rede pode garantir que a rede tenha capacidade de acomodar os usuários e aplicações necessárias. Exemplos de métricas são: taxa de serviço, disponibilidade e utilização de recursos, tempo de resposta e taxas de erro.

- *Gerência de configuração* diz respeito a inicialização e manutenção de dispositivos e aplicações dentro de uma rede. Um exemplo de gerência de configuração é controlar a versão dos *softwares* instalados em cada máquina.

- *Gerência de contas* envolve monitorar a utilização de recursos da rede por cada usuário ou grupo de usuários para melhor garantir que os usuários tenham recursos suficientes, através do estabelecimento de cotas, determinação de custos ou contagem de usuários. Isto também envolve conceder ou remover permissões de acesso à rede.

- *Gerência de segurança* é o processo de controlar acesso aos dados armazenados em uma rede, controlar o acesso a recursos ou controlar e monitorar dispositivos de segurança, como *firewalls*.

A necessidade de controlar e monitorar redes cada vez mais extensas e complexas, com componentes heterogêneos, levou ao uso de ferramentas automáticas para gerência de redes. É necessário padronizar ferramentas de gerência, a fim de que a comunicação entre dispositivos diferentes, e de diversos fabricantes, possa ser mantida. Em resposta a esta necessidade, diversos padrões foram desenvolvidos. O *Simple Network Management Protocol* (SNMP) é o protocolo de gerência de redes TCP/IP padrão da Internet.

SNMP é um conjunto de padrões para gerência de redes, incluindo um protocolo, uma especificação para estrutura de armazenamento de dados, e um conjunto de objetos de dados. O SNMP encontra-se em sua terceira versão (SNMPv3) [HPW99]. Há atualmente um grande número de sistemas baseados no SNMP, tanto comerciais quanto de domínio público [ucd, snm, sim]. Muitos dos dispositivos mencionados no início desta seção já vêm preparados para receber *software* compatível com SNMP.

Uma característica do SNMP é seguir o axioma fundamental da gerência, onde "o impacto de adicionar a gerência de redes aos dispositivos gerenciados deve ser mínimo" [Ros94]. O SNMP é projetado para ser fácil de implementar e para consumir o mínimo de processamento e uso dos recursos da rede.

O original sistema de gerência de redes padrão da Internet (SNMPv1) [CFSD90] descrevia

uma linguagem para definição de dados, informação de gerência, as operações do protocolo e uma abordagem simples para segurança e administração. A segunda versão do SNMP (SNMPv2) [CMRW96b] trouxe diversas vantagens, desde novos tipos de dados até novas operações mais eficientes, incluindo uma consistente indicação de erros. A última versão (SNMPv3) [HPW99] acrescentou um elaborado sistema de segurança e de controle de acesso aos dados, além de suportar interação com as outras duas versões.

A seguir, será apresentada a estrutura de um sistema de gerência de redes baseado no SNMP e cada uma de suas partes será detalhada. Por último, será mostrado como estes componentes interagem a fim de realizar a tarefa de controle e monitoramento dos dispositivos de uma rede de computadores.

3.2 Sistema de Gerência de Redes Baseado no SNMP

Um sistema de gerência de redes baseado no SNMP [HPW99, Ros94] contém:

- um conjunto de *entidades* distribuídas que interagem entre si fazendo uso do protocolo de gerência;
- um *protocolo de gerência*, que define *operações* usadas para transmitir informações de gerência;
- *informações de gerência*, que são as informações coletadas nos dispositivos gerenciados.

Uma entidade SNMP podem atuar, basicamente, em dois papéis, ou numa combinação destes dois: como *gerente*, quando tem a função de controlar e monitorar os dispositivos gerenciados através da manipulação das informações de gerência; ou como *agente*, quando tem a função de tornar os dispositivos gerenciados acessíveis através do armazenamento das informações coletadas nestes ambientes.

O protocolo de gerência disponibiliza operações que permitem que uma entidade possa consultar (monitorar) e alterar (controlar) as informações de gerência dos dispositivos gerenciados, através do mecanismo de *polling*. Além disto, através do mecanismo de alarmes, define operações que permitem que entidades possam notificar-se sobre um evento extraordinário que tenha ocorrido.

As informações de gerência são referentes aos dispositivos gerenciados, podendo ser, por exemplo, dados sobre o número de pacotes recebidos por um dispositivo, ou se o estado do dispositivo está falho ou sem falha. Cada unidade de informação é tratada como um *objeto de gerência*.

As informações de gerência de um dispositivo são armazenadas numa entidade com função de agente, localizada no próprio dispositivo. Outras entidades na função de gerente podem alterar ou consultar estas informações, desde que tenham acesso para isto. A comunicação entre as entidades se estabelece através da troca de mensagens contendo as informações de gerência e o tipo de operação a ser executada.

Um exemplo de sistema de gerência de redes baseado no SNMP pode ser visto na Figura 3.1. Entidades no papel de gerente ou de agente interagem entre si através do protocolo de gerência, trocando pacotes de dados. As informações de gerência ficam armazenadas nas entidades atuando como agentes e são manipuladas pelas entidades atuando como gerentes.

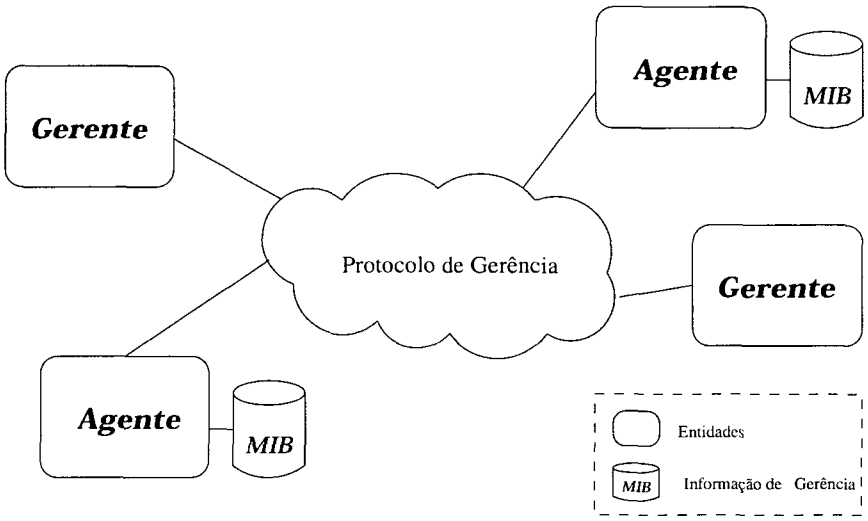

Figura 3.1: Sistema de Gerência de Redes baseado no SNMP.

A seguir, cada componente de um sistema de gerência de redes baseado no SNMP será tratado com maiores detalhes.

3.3 Informações de Gerência

Como foi visto anteriormente, um item de informação de gerência é visto como um *objeto* gerenciado. Objetos podem ser *simples*, quando são acessados como uma única informação, ou *agregados*, quando formam tabelas. Coleções de objetos relacionados são armazenados em estruturas de armazenamento virtuais denominadas *Management Information Bases* (MIBs) [CMPS99]. Usualmente, uma MIB contém definições de objetos de gerência, mas pode também conter definições de notificações de evento.

Abstract Syntax Notation One (ASN.1) é uma linguagem formal desenvolvida e padronizada por CCITT (X.208) e ISO (ISO 8824). ASN.1 é importante por diversas razões, incluindo a

Objeto	Descrição	Sintaxe
snmpTrapOID	Identificação da notificação atualmente enviada	*OBJECT IDENTIFIER*
sysName	Nome do dispositivo gerenciado	*String*
sysDescr	Descrição textual do dispositivo	*String*
sysUpTime	O tempo desde que o sistema foi inicializado	*TimeTicks*

Tabela 3.1: Exemplos de objetos de gerência do protocolo SNMPv3.

definição de sintaxe abstrata dos dados das aplicações e a estrutura das mesmas [Sta98b].

As informações de gerência devem ser independentes da versão e da implementação do SNMP. Para tanto, estas MIBs são escritas utilizando-se um subconjunto da ASN.1 [fS87, Sta98b], chamado de *Structure of Management Information* (SMI), agora em sua segunda versão [MPS99b, MPS99c, MPS99a], que estabelece a sintaxe para se descrever informação de gerência. SMI define tipos de dados fundamentais, modelo de objetos de gerência, e regras para escrever e revisar módulos MIB.

Alguns exemplos de objetos de gerência podem ser vistos na Tabela 3.1

Faz-se necessário também definir alguns termos constantemente utilizados nesta literatura:

- o termo *variable* (ou *variável*) refere-se a uma instância de um objeto não agregado definido de acordo com as convenções estabelecidas pelo SMI;
- o termo *variable-binding* refere-se a um par formado pelo nome de uma variável e pelo valor da mesma variável. Este par é usado para acessar ou alterar uma informação de gerência numa MIB. Entretanto, se uma condição excepcional ocorrer durante o processamento de uma requisição recuperando uma informação de gerência, o par poderá conter o nome da variável e uma indicação para esta condição excepcional.
- o termo *variable-binding-list* é uma lista contendo *variable-bindings*.

3.4 Operações do Protocolo

O protocolo de gerência do SNMP [CMRW96b] provê a troca de informações de gerência através do envio e recepção de *Protocol Data Units* (PDUs), que são os pacotes trocados entre as entidades SNMP. Um PDU indica o tipo de operação de gerência e a lista de nomes de variáveis relacionadas a esta operação.

Um PDU é composto por: um campo *"operation-type"* para indicar a operação do protocolo, um campo *"request-id"* que é o índice do PDU enviado, um campo *"variable-bindings"* contendo

Operação	Descrição	PDU
Get	Consulta um objeto de gerência	GetRequest
GetNext	Consulta o próximo objeto de gerência	GetNextRequest
GetBulk	Consulta um conjunto de objetos de gerência	GetBulkRequest
Set	Altera um objeto de gerência	SetRequest
Inform	Propaga objetos de gerência entre entidades	InformRequest
Trap	Alerta uma entidade sobre um evento extraordinário	Trapv2
Response	Resposta às operações descritas acima	Response
Report	Notificação interna à entidade	Report

Tabela 3.2: Operações do protocolo SNMPv3.

uma lista de *variable-bindings*, um campo chamado *"error-status"* para indicar uma mensagem de erro e outro chamado *"error-index"* para indicar o índice da variável, na lista, na qual o erro ocorreu. O campo *"operation-type"* diferencia os diversos PDUs. SMI também é usado para definir o formato dos PDUs trocados pelas entidades.

Uma descrição informal das operações do SNMPv3 e seus respectivos PDUs pode ser vista na Tabela 3.2.

Os PDUs podem ser classificados pelas propriedades funcionais que exercem [CMPS99], como:

- *de leitura*, quando contêm operações que recuperam informação de gerência (GetRequest-PDU, GetNextRequest-PDU, GetBulkRequest-PDU);
- *de escrita*, quando contêm operações que modificam informação de gerência (SetRequest-PDU);
- *de notificação*, quando contêm operações que notificam outras entidades sobre eventos extraordinários, funcionando como sinais de alerta (InformRequest-PDU, Trapv2-PDU);
- *de resposta*, quando contêm operações que são enviadas em resposta a requisições anteriores (Response-PDU);
- *interno*, quando contêm operações que são trocadas internamente entre engenhos SNMP[1] (Report-PDU).

As operações comumente chamadas de *requisições* são todas estas citadas acima, com exceção da operação de resposta.

[1]Engenhos SNMP são apresentados na Seção 3.5.

Independentemente do seu tipo, um PDU é encapsulado em uma *mensagem* SNMP antes de ser transmitido entre entidades. Esta mensagem carrega informações de segurança e controle de acesso dos dados requisitados. Uma mensagem diferente é definida para cada versão do SNMP.

Para maiores informações sobre o formato dos PDUs, refira-se a [CFSD90, CMRW96b]. Para uma descrição sobre o formato da mensagem do SNMPv3, veja [CHPW99, Sta98a].

3.4.1 Papel das Entidades Segundo sua Funcionalidade

Como mostrado anteriormente, uma entidade pode atuar no papel de gerente ou de agente (Seção 3.2). Seu papel também pode ser definido através das operações que manipula, como mostrado a seguir [CMRW96b]:

1. Uma entidade atua no *papel de gerente* quando gera requisições e notificações, processando sua resposta (envia GetRequest-PDU, GetNextRequest-PDU, GetBulkRequest-PDU, SetRequest-PDU, InformRequest-PDU; e então recebe Response-PDU) e quando processa notificações, gerando sua resposta quando necessário (recebe InformRequest-PDU, Trapv2-PDU; e então envia Response-PDU).

2. Uma entidade atua no *papel de agente* quando processa requisições, gerando sua resposta recebe GetRequest-PDU, GetNextRequest-PDU, GetBulkRequest-PDU, SetRequest-PDU; e então envia Response-PDU) e quando gera notificações, sem processar nenhuma resposta (envia Trapv2-PDU).

Uma mesma entidade SNMP pode suportar um ou ambos os papéis simultaneamente.

3.4.2 Interações entre Entidades Segundo sua Funcionalidade

Três tipos de interações são providas entre as entidades SNMP, baseado no papel que estas entidades exercem [CMRW96b], como pode ser visto na Figura 3.2. No primeiro tipo, entidades atuando como gerentes enviam requisições para entidades atuando como agentes e recebem a resposta destes. No segundo tipo, entidades atuando como gerentes enviam notificações para entidades atuando como gerentes e recebem a resposta destes. No terceiro tipo, entidades atuando como agentes enviam notificações para entidades atuando como gerentes, sem esperar resposta destes.

3.5 Estrutura das Entidades

Uma entidade SNMP possui a seguinte composição interna [HPW99, Sta98c]:

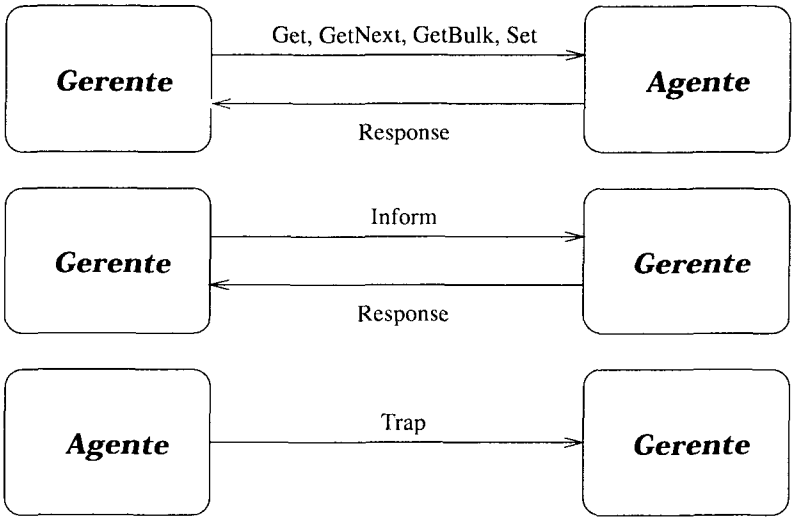

Figura 3.2: Relação entre operações e papel das entidades.

- uma ou mais *aplicações SNMP*, que são os módulos que irão executar as operações de gerência. Cada categoria de aplicação tem uma finalidade e suporta certos tipos de operações do protocolo. Elas coordenam o processamento das operações sobre informações de gerência;
- uma única plataforma, denominada *engenho SNMP*, provê serviços para as aplicações, tais como montar e enviar mensagens, receber e desmontar mensagens, fazer autenticação e criptografia dos dados, e controle de acesso aos objetos gerenciados.

A seguir, cada um dos módulos da entidade será detalhado. A interação entre cada parte será mostrada no final da seção. Além dos componentes aqui apresentados, novos módulos podem ser especificados conforme a necessidade, devido à propriedade modular desta arquitetura [HPW99].

3.5.1 Engenho

Um *Engenho SNMP* contém os seguintes módulos componentes: um despachante, um subsistema de processamento de mensagens, um subsistema de segurança e um subsistema de controle de acesso.

O *Despachante* [CHPW99] é o módulo responsável por controlar o tráfego dentro da entidade, enviando e recebendo mensagens SNMP, além de prover uma interface abstrata para as aplicações SNMP se comunicarem com outras entidades SNMP ou com outros programas.

A seguir, será mostrado os procedimentos que podem ser executados por um Despachante:

1. Para PDUs deixando a entidade, o Despachante aceita um PDU de uma aplicação e determina o tipo de processamento de mensagem requerido (i.e., SNMPv1, SNMPv2, SNMPv3)

e passa o PDU para o modelo de processamento de mensagens apropriado. Subsequentemente, o Subsistema de Processamento de Mensagens retorna uma mensagem contendo aquele PDU e incluindo um cabeçalho apropriado à mensagem. O Despachante então direciona esta mensagem à camada de transporte para transmissão.

2. Para mensagens chegando na entidade, o Despachante aceita mensagens provindas da camada de transporte e determina a sua versão para enviá-lo ao modelo de processamento de mensagem apropriado. Subsequentemente, o Subsistema de Processamento de Mensagens retorna o PDU contido na mensagem, o qual é passado à aplicação apropriada.

O *Subsistema de Processamento de Mensagens* é o módulo responsável pelos serviços de empacotamento de PDUs em mensagens de diferentes versões do SNMP, assim como de extração de PDUs contidos em mensagens recebidas. Uma implementação deste módulo deve suportar um ou mais modelos distintos de processamento de mensagem. Um modelo para a mensagem do SNMPv3 (v3MP) está definido em [CHPW99].

O *Subsistema de Segurança* é o módulo que provê serviços de autenticação e criptografia dos dados. Para mensagens a serem enviadas, dependendo dos serviços requeridos, o Subsistema de Segurança pode criptografar o PDU e alguns campos do cabeçalho da mensagem e pode gerar um código de autenticação e inserí-lo no cabeçalho da mensagem. A mensagem processada é então retornada ao Subsistema de Processamento de Mensagens. Para mensagens recebidas, se requerido, o Subsistema de Segurança verifica o código de autenticação e executa a decriptografia e então retorna a mensagem para o Subsistema de Processamento de Mensagens. O único modelo de segurança existente atualmente, o *User-based Security Model* (USM), está definido em [BW99].

O *Subsistema de Controle de Acesso* é o módulo que provê serviços de autorização de acesso aos objetos de gerência. Esta autorização pode ser para leitura e escrita nos ditos objetos. Estes serviços são executados conforme o conteúdo dos PDUs. O único modelo de controle de acesso existente atualmente, o *View-based Access Control Model* (VACM), está definido em [WPM99].

Um Engenho SNMP deve ser único dentro de uma entidade. Além disto, cada engenho deve conter somente um módulo Despachante, para que não haja conflito no envio e no recebimento de mensagens.

3.5.2 Aplicações

As seguintes aplicações SNMP [LMS99] são aceitas por um engenho SNMP: Gerador de Comandos, Receptor de Comandos, Gerador de Notificações, Receptor de Notificações e *Proxy*

Forwarder. Elas são descritas resumidamente a seguir.

Uma aplicação *Gerador de Comandos* (*Command Generator*) monitora e manipula informações de gerência localizadas em agentes remotos através da geração de requisições (GetRequest-PDU, GetNextRequest-PDU, GetBulkRequest-PDU, SetRequest-PDU) e processando as respostas (Response-PDU) para os pedidos que iniciou.

Uma aplicação *Receptor de Comandos* (*Command Responder*) provê acesso às informações de gerência através do recebimento de requisições (GetRequest-PDU, GetNextRequest-PDU, GetBulkRequest-PDU, SetRequest-PDU) e gera resposta (Response-PDU) aos pedidos que recebeu após recuperar ou alterar objetos de gerência.

Uma aplicação *Gerador de Notificações* (*Notification Originator*) monitora um sistema por eventos específicos ou condições pré-determinadas gerando mensagens de alerta ou de notificação (InformRequest-PDU, Trapv2-PDU) baseadas nestes eventos ou condições. A resposta é processada (Response-PDU) quando um pacote do tipo InformRequest-PDU tiver sido gerado.

Uma aplicação *Receptor de Notificações* (*Notification Receiver*) processa mensagens de notificação (InformRequest-PDU, Trapv2-PDU). No caso de recepção de um InformRequest-PDU, a aplicação gera uma resposta (Response-PDU).

Uma aplicação Proxy Forwarder propaga mensagens entre entidades e sua implementação é opcional [LMS99, Seção 1.5].

3.5.3 Papel das Entidades Segundo sua Estrutura

Na Seção 3.4, viu-se como caracterizar o papel de uma entidade SNMP pelas operações que ela manipula. Uma entidade também pode ter seu papel determinado pelas aplicações que possui. Esta é uma das vantagens da abordagem modular para construção de entidades, proposto em [HPW99].

1. Uma entidade SNMP atua no *papel de gerente* quando possui três categorias de aplicações: Gerador de Comandos, Gerador de Notificações (para enviar somente InformRequest-PDU) e Receptor de Notificações. Além disto, seu engenho contém um Despachante, um Subsistema de Processamento de Mensagens e um Subsistema de Segurança. A Figura 3.3 mostra a arquitetura tradicional de um gerente SNMP.

2. Uma entidade SNMP atua no *papel de agente* quando possui três categorias de aplicações: Receptor de Comandos, Gerador de Notificações (para enviar somente Trapv2-PDU) e *Proxy Forwarder*. Além disto, seu engenho contém um Despachante, um Subsistema de Processamento de Mensagens, um Subsistema de Segurança e um Subsistema de Controle

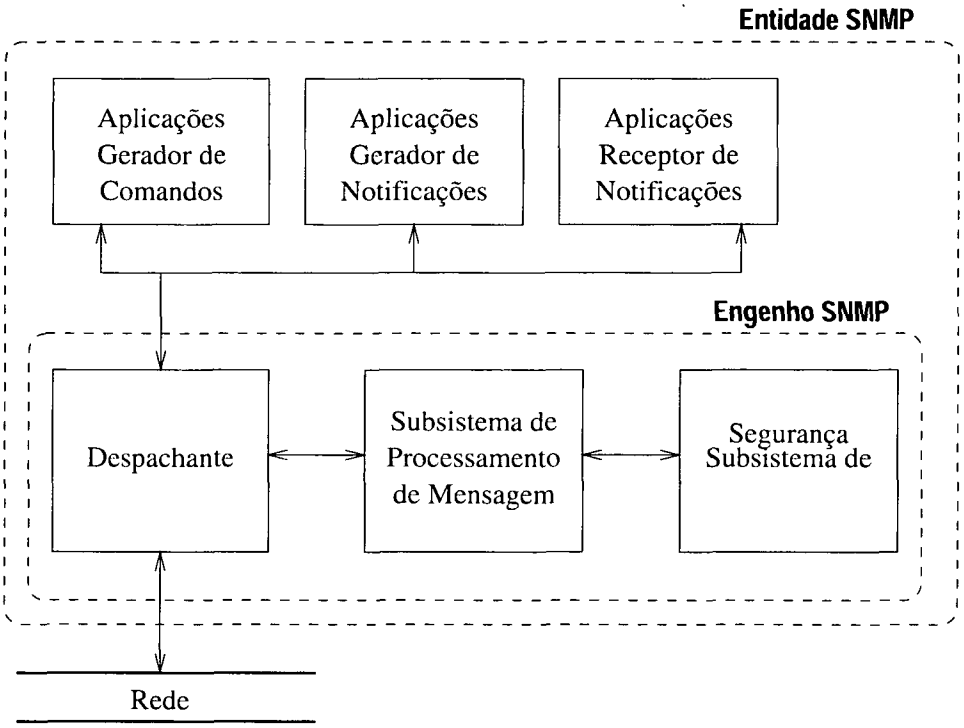

Figura 3.3: Gerente SNMP.

de Acesso. Além disto, aplicações Receptor de Comandos podem acessar a Instrumentalização das MIBs, que consiste nas MIBs e nos comandos para consultar ou alterar os seus objetos. A Figura 3.4 mostra a arquitetura tradicional de um agente SNMP.

Novos papéis podem ser atribuídos a uma entidade, dependendo da variação de sua estrutura e funcionalidade (quais operações manipula).

3.5.4 Interações entre Componentes de uma Entidade

Os serviços providos entre módulos em uma entidade SNMP [HPW99] estão definidos em termos de primitivas e parâmetros. As primitivas especificam a função a ser desempenhada e os parâmetros são usados para passar dados e informação de controle entre os módulos.

A Figura 3.5 mostra as primitivas utilizadas, e a ordem de chamada entre elas, no envio de uma requisição, e na sua recepção por outra entidade. Na entidade de origem, uma aplicação Gerador de Comandos ou Gerador de Notificações usa a primitiva *sendPdu* para se comunicar com o Despachante da entidade. O Despachante usa a primitiva *prepareOutgoingMsg* para se comunicar com o Subsistema de Processamento de Mensagens da entidade. O Subsistema de Processamento de Mensagens usa a primitiva *generateRequestMsg* para se comunicar com o Subsistema de Segurança da entidade. Então, quando o Despachante já obteve a resposta da

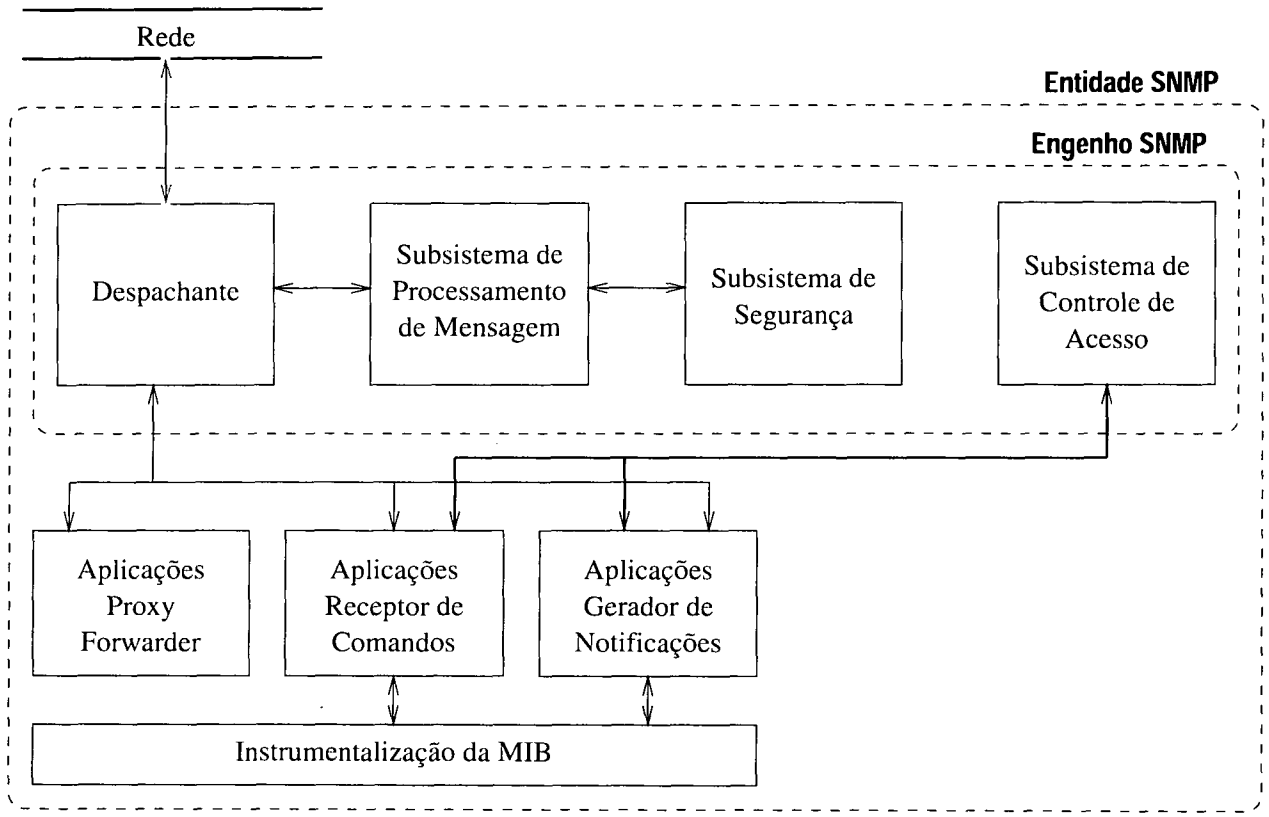

Figura 3.4: Agente SNMP.

primitiva *prepareOutgoingMsg*, ele envia o *NetworkMsg* recebido para outra entidade. Na entidade de destino, o seu Despachante usa a primitiva *prepareDataElements* para se comunicar com o Subsistema de Processamento de Mensagens. O Subsistema de Processamento de Mensagens usa a primitiva *processIncomingMsg* para se comunicar com o Subsistema de Segurança. Então, com a mensagem recebida já processada, o Despachante usa a primitiva *processPdu* para enviar a operação de gerência de redes para uma aplicação Receptor de Comandos ou Receptor de Notificações.

A Figura 3.6 mostra as primitivas utilizadas para o envio de uma resposta e a sua recepção na entidade de origem. Na entidade de origem, uma aplicação Receptor de Comandos ou Receptor de Notificações usa a primitiva *returnResponsePdu* para se comunicar com o Despachante da entidade. O Despachante usa a primitiva *processResponseMsg* para se comunicar com o Subsistema de Processamento de Mensagens da entidade. O Subsistema de Processamento de Mensagens usa a primitiva *generateResponseMsg* para se comunicar com o Subsistema de Segurança da entidade. Então, quando o Despachante já obteve a resposta da primitiva *processResponseMsg*, ele envia o *NetworkMsg* recebido para outra entidade. Na entidade de destino, o seu Despachante usa a primitiva *prepareDataElements* para se comunicar com o Subsistema de Processamento de

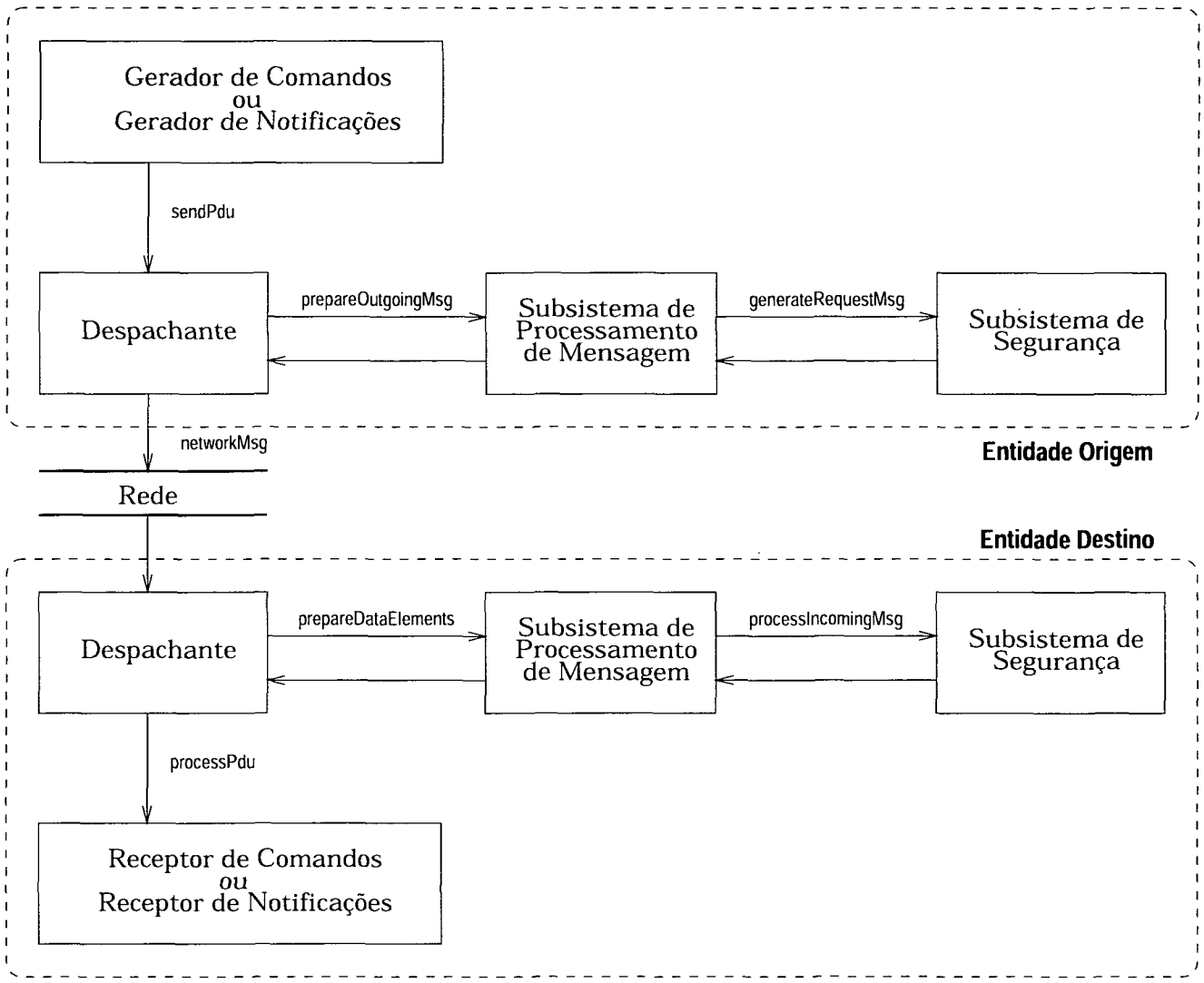

Figura 3.5: Processo de requisição de uma operação SNMP.

Mensagens. O Subsistema de Processamento de Mensagens usa a primitiva *processIncomingMsg* para se comunicar com o Subsistema de Segurança. Então, com a mensagem recebida já processada, o Despachante usa a primitiva *processResponsePdu* para enviar a resposta da operação de gerência de redes para uma aplicação Gerador de Comandos ou Gerador de Notificações.

Uma vez enviada uma requisição, sua resposta é esperada até que chegue, de forma síncrona. Já uma requisição é recebida de forma assíncrona, pois nunca se sabe quando ela chegará.

3.5.5 Interações entre Entidades Segundo sua Estrutura

Três tipos de interações são providas entre as entidades SNMP, dependendo do papel que estas entidades exercem [CMRW96b], como podem ser vistas na Figura 3.7. No primeiro tipo, uma aplicação Gerador de Comandos de uma entidade atuando como gerente envia uma requisição

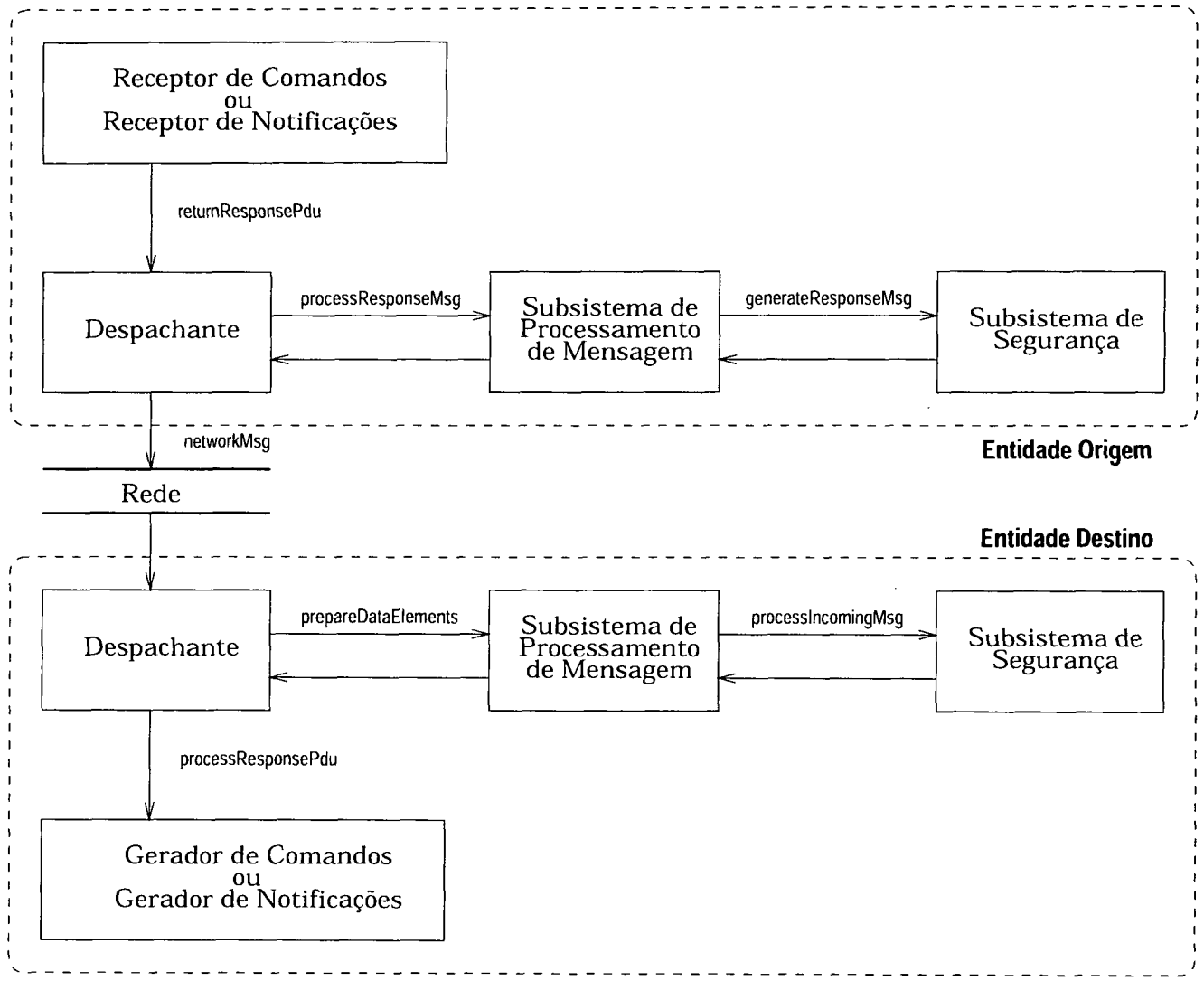

Figura 3.6: Processo de resposta de uma operação SNMP.

para uma aplicação Receptor de Comandos de uma entidade atuando como agente e recebe a resposta desta. No segundo tipo, uma aplicação Gerador de Notificações de uma entidade atuando como gerente envia uma notificação para uma aplicação Receptor de Notificações de uma entidade atuando como gerente e recebe a resposta desta. No terceiro tipo, uma aplicação Gerador de Notificações de uma entidade atuando como agente envia uma notificação para uma aplicação Receptor de Notificações de uma entidade atuando como gerente, sem esperar resposta desta.

Em todos os casos, são usadas as primitivas descritas na seção anterior para se fazer as trocas de mensagens entre as entidades.

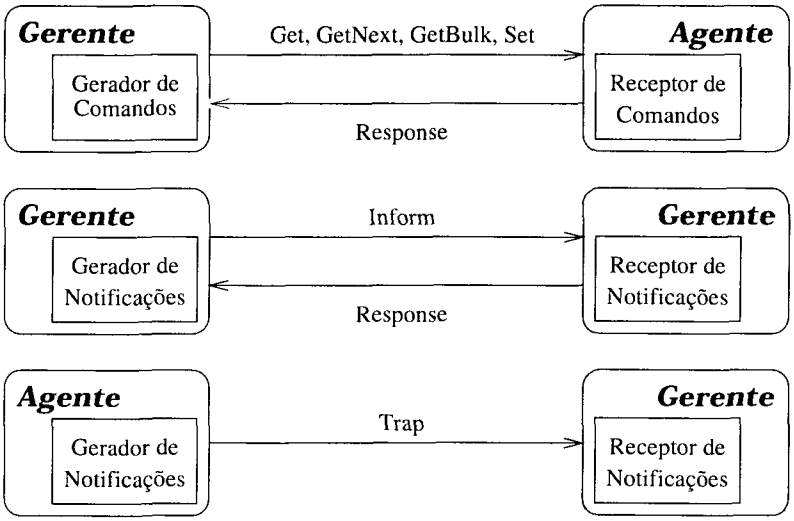

Figura 3.7: Relação entre aplicações e papel das entidades.

3.6 Considerações Finais

Neste capítulo introduzimos os conceitos mais importantes de SNMP, que incluem os tipos de entidades, módulos, componentes, operações e primitivas utilizadas.

Nos capítulos posteriores, uma entidade SNMP será especificada formalmente usando semântica de ações, de forma que se tenha agentes da semântica de ações executando ações que correspondem aos procedimentos seguidos pelos módulos de uma entidade SNMP.

Para obter informações históricas sobre o SNMP, refira-se a [CMPS99, Sta98c]. Para obter detalhes sobre SNMPv1, veja [CFSD90]. Para obter detalhes sobre SNMPv2, veja [CMRW96b].

Para ter acesso a implementações do SNMP, refira-se a [ucd, snm, sim].

CAPÍTULO 4

Trabalhos Correlatos

Este capítulo traz os trabalhos correlatos a esta dissertação, que envolvem os trabalhos que antecederam a este, além de especificações de outras aplicações usando semântica de ações. Existe uma ausência de trabalhos na literatura com respeito a especificações formais de protocolos de gerência de redes.

Em [Mus92], foi proposto o uso de semântica de ações para a descrição formal do *Sun Remote Procedure Call Protocol* que é considerado como uma base para a atual implementação de *Remote Procedure Call* (RPC) em muitos sistemas de computação distribuída.

Em [DM99], propõe-se o uso de semântica de ações para especificar o comportamento de objetos de gerência de redes. Isto inclui a especificação do mecanismo de leitura e escrita para cada objeto de gerência, que pode ser simplesmente adicionado ao campo *Description* da descrição já existente do objeto de gerência em ASN.1, sem implicar em modificações nos padrões atuais. Inicialmente é definida a semântica de um agente SNMP primitivo e das aplicações snmpGet e snmpSet, e então é feito um estudo de caso, onde define-se a semântica de uma MIB experimental denominada *SNMP Routing Proxy MIB*. O trabalho [DM99] serviu de base para a descrição semântica de entidades de gerência de redes introduzida nesta dissertação.

Em [FMD00a], apresenta-se uma descrição formal das aplicações padrão do SNMP usando semântica de ações. Como será visto neste trabalho, as aplicações padrão fazem parte de uma entidade SNMP e são responsáveis por emitir e processar operações de gerência.

Em [FMD00b], apresenta-se uma descrição formal do Despachante SNMP usando semântica de ações. Como será visto neste trabalho, o Despachante é um módulo da entidade SNMP, sendo responsável por controlar o fluxo de mensagens dentro de uma entidade e entre entidades.

Embora exista na literatura um grande número de especificações formais de protocolos de comunicação, é importante observar que não existem especificações formais do SNMP.

CAPÍTULO 5

Operações SNMP

Este capítulo trata da definição formal das operações SNMP. A Seção 5.1 traz a conceituação de uma entidade primitiva para servir de motivação inicial no uso das operações SNMP, que são definidas formalmente na Seção 5.2.1, englobando sua solicitação e a resposta das mesmas. Operações auxiliares, abrangendo variáveis do sorte *VarBind* são definidas na Seção 5.2.2, e operações para manipular objetos de uma MIB são definidas na Seção 5.3. Na Seção 5.4, a notação auxiliar é apresentada, contendo dados, produtores e ações usados neste capítulo.

O trabalho exposto neste capítulo é uma versão mais realista e atualizada de [DM99]. Parte do trabalho aqui exposto já foi mostrado em [FMD00a]. Toda a especificação a seguir é baseada na descrição informal das operações SNMP em [CMRW96b].

5.1 Entidade Primitiva

Para especificar as operações SNMP, faz-se necessário introduzir um mecanismo para sua manipulação. Nesta seção é definida uma entidade SNMP primitiva, que usa uma arquitetura simplificada para enviar e receber PDUs, ao invés de mensagens mais complexas. Esta entidade se baseia na apresentada em [DM99], não possuindo engenho, nem aplicações padrão. Ela se caracteriza por conter apenas uma aplicação, que exerce a função de gerente ou de agente, e faz uso das operações SNMP no processo de gerar requisições ou de produzir sua resposta.

Desta forma, quer-se isolar a especificação das operações SNMP em si, que consistem em transmitir (formar e processar) PDUs, sem considerar a mensagem SNMP transmitida entre entidades para prover serviços para as aplicações, incluindo o processamento do tipo da versão da mensagem, criptografia dos dados e o controle de qual entidade tem acesso a quais dados.

Esta abordagem também possibilita uma visão *top-down* da especificação das operações SNMP. A mensagem SNMP com seus serviços será então abordada nos capítulos que seguem.

5.1.1 Requisições

As ações apresentadas a seguir são executadas quando uma aplicação necessita iniciar uma operação SNMP. Existe uma ação para cada operação SNMP.

- snmpGet(_, _) :: entity, list of ObjectName$^+$ → action
- snmpGetNext(_, _) :: entity, list of ObjectName$^+$ → action
- snmpGetBulk(_, _, _, _) :: entity, Index, Index, list of ObjectName$^+$ → action
- snmpSet(_, _) :: entity, VarBindList → action
- snmpInform(_, _) :: entity, Notification-Macro → action
- snmpTrap(_, _) :: entity, Notification-Macro → action

(1) snmpGet(E:entity, N:list of ObjectName$^+$) =
serviceRequest Get to E with (0, 0, N)

(2) snmpGetNext(E:entity, N:list of ObjectName$^+$) =
serviceRequest GetNext to E with (0, 0, N)

(3) snmpGetBulk(E:entity, R:Index, M:Index, N:list of ObjectName$^+$) =
serviceRequest GetBullk to E with (R, M, N)

(4) snmpSet(E:entity, V:VarBindList) =
serviceRequest Set to E with (0, 0, V)

(5) snmpInform(E:entity, M:Notification-Macro) =
serviceRequest Inform to E with (0, 0, M)

(6) snmpTrap(E:entity, M:Notification-Macro) =
serviceRequestWOResponse Trap to E with (0, 0, M)

As ações definidas acima são responsáveis por propiciar uma interface de utilização para as operações SNMP. As ações snmpGet, snmpGetNext e snmpGetBulk recebem uma lista de nomes de objetos de gerência a serem consultados (variável N). A ação snmpSet recebe uma *VarBindList* já formada, com os valores a serem alterados (variável V). Já as ações snmpInform e snmpTrap recebem o nome da macro de notificação que contém os objetos a serem enviados na notificação (variável M).

Além disto, a ação snmpGetBulk precisa dos valores de *"non-repeaters"* e *"max-repetitions"*, que são passados pelas variáveis R e M. Em todas as outras operações estes valores não são inicializados [CMRW96b], sendo então atribuídos para 0.

Todas estas ações executam serviceRequest, que é a ação que representa o processo de envio de uma requisição e a espera de sua resposta. Quando uma resposta não precisa ser aguardada, a ação serviceRequestWOResponse é executada. A ação serviceRequest é especificada a seguir:

- serviceRequest _ to _ with (_, _, _) :: opTag, agent, Index, Index, tuple → action

(7) serviceRequest Op:opTag to E:entity with (S:Index, I:Index, T:tuple) =
| | operationRequest Op with (S, I, T)
then
| | send a message[to E][containing the given PDU]
and then
| | accept contents of a message[from E][containing a PDU tagged with Response]
then
| | give (the error-status of the given PDU, the error-index of the given PDU,
| | the variable-bindings of the given PDU)

Esta ação recebe cinco parâmetros: (1) E é a entidade para a qual se destina a operação; (2) Op é a operação requisitada; (3) S e (4) I são respectivamente os valores dos campos *"max-repetitions"* e *"non-repeaters"* do PDU e (5) T é uma tupla que será processada para fornecer os objetos de gerência que devem ser operados. Todos estes parâmetros serão utilizados para formar o PDU a ser enviado.

Inicialmente é executada operationRequest, definida na Seção 5.2.1, que é a ação que produz o PDU a ser enviado, a partir dos dados S, I e T fornecidos para ela. Em seguida, este PDU é enviado para a entidade indicada por E e um PDU de resposta é esperado desta mesma entidade. Então, este PDU de resposta é desempacotado e os valores de *"error-status"*, *"error-index"* e *"variable-bindings"* são retornados para a aplicação que executou a operação.

A especificação de serviceRequestWOResponse é similar à especificação de serviceRequest. Entretanto, nenhuma mensagem de resposta é esperada ou retornada.

- serviceRequestWOResponse _ to _ with (_, _, _) :: opTag, agent, Index, Index, tuple → action

(8) serviceRequestWOResponse Op:opTag to E:entity with (S:Index, I:Index, T:tuple) =
| *operationRequest* Op with (S, I, T)
then
| send a message[to E][containing the given PDU]

5.1.2 Respostas

As ações apresentadas a seguir são executadas por uma entidade quando é necessário responder a operações SNMP recebidas. Para este procedimento, duas ações são definidas: Agent-Daemon e Manager-Daemon, correspondente ao agente e gerente SNMP, respectivamente.

Uma entidade primitiva no papel de agente SNMP, quando respondendo a operações de gerência, vai executar a ação Agent-Daemon, definida a seguir:

- Agent-Daemon :: action

(9) Agent-Daemon =
| initialize LCD-Primitive
hence unfolding
| | | accept a message[from any entity][containing a PDU]
| | then
| | | serviceResponse Get
| | or
| | | serviceResponse GetNext
| | or
| | | serviceResponse GetBulk
| | or
| | | serviceResponse Set
| and then unfold

Esta ação simplesmente aceita uma mensagem e inicia os serviços seguidos por um agente SNMP. Caso o PDU recebido contenha uma das operações processáveis pela entidade, ela será detectada e executada dentro da ação serviceResponse. Caso contrário, uma nova mensagem é esperada.

A ação initialize LCD-Primitive[1] é responsável por prover o ambiente de definição que será utilizado para salvar informações auxiliares, chamado *Local Configuration Datastore* (LCD), durante o processo de resposta de uma operação SNMP, antes que qualquer mensagem seja esperada ou processada. Os seguintes campos são necessários na LCD de uma entidade primitiva: *"sender"*, *"request-id"*, *"non-repeaters"*, *"max-repetitions"* e *"variable-bindings"*.

```
• initialize LCD-Primitive :: action
(10) initialize LCD-Primitive =
        | allocate a cell then bind "sender" to it
       and
        | allocate a cell then bind "request-id" to it
       and
        | allocate a cell then bind "non-repeaters" to it
       and
        | allocate a cell then bind "max-repetitions" to it
       and
        | allocate a cell then bind "variable-bindings" to it
```

A ação accept[2] recebe uma mensagem, e então o seu remetente é salvo com o rótulo *"sender"*, e o seu conteúdo é retornado como informação transitória.

Uma entidade primitiva no papel de gerente SNMP, quando respondendo a operações de gerência, vai executar a ação Manager-Daemon, definida a seguir:

```
• Manager-Daemon :: action
(11) Manager-Daemon =
        | initialize LCD-Primitive
       hence unfolding
        | | | accept a message[from any entity][containing a PDU]
        | | then
        | | | | serviceResponse Inform
        | | or
        | | | | serviceResponseWOResponse Trap
        | and then unfold
```

Esta ação simplesmente aceita uma mensagem e inicia os serviços seguidos por um gerente SNMP. Caso o PDU recebido contenha uma das operações processáveis pela entidade, ela será detectada e executada dentro da ação serviceResponse ou serviceResponseWOResponse. Caso contrário, uma nova mensagem é esperada.

A ação serviceResponse representa o processo de recepção de uma requisição e o posterior envio de sua resposta:

```
• serviceResponse _ :: opTag → action[receiving PDU]
(12) serviceResponse Op:opTag =
```

[1]Definida na Seção 5.4.

[2]Definida na Seção 5.4.

```
    | | give the given PDU tagged with Op
    | then
    | | give (request-id of it, error-status of it, error-index of it, variable-bindings of it)
    | then
    | | cache the splitted PDU
  and then
    | | operationResponse Op
    | then
    | | give the PDU of (the cached "request-id", the given error-status#1,
    | |                  the given error-index#2, the given VarBindList#3) tagged with Response
    | then
    | | send a message[to the cached "sender"][containing the given PDU]
```

Esta ação recebe um PDU como informação transitória, que será validado pelo produtor the given PDU tagged with Op[3], que irá completar somente se o campo *"operation-type"* do PDU recebido tiver o mesmo valor que *Op* (note que a ação falha quando o PDU fornecido a ela não for do sorte *Op*). Caso isto proceda, o PDU é desmontado e seus campos são salvos para posterior utilização pela ação cache the splitted PDU[4] na memória alocada pela ação initialize LCD-Primitive.

Após a operação de gerência ser tratada, um PDU é construído com o rótulo *Response* e com os dados por ela fornecido, e é então enviado. A referência para o agente que enviou a requisição é guardada durante a recepção da mensagem, pois será utilizada para o envio da mensagem de resposta no final da ação serviceResponse.

A especificação de serviceResponseWOResponse é similar à especificação de serviceResponse. Entretanto, nenhuma mensagem de resposta é gerada ou enviada.

```
• serviceResponseWOResponse _ :: opTag → action[receiving PDU]
(13) serviceResponseWOResponse Op:opTag =
       | | give the given PDU tagged with Op
       | then
       | | give (request-id of it, error-status of it, error-index of it, variable-bindings of it)
       | then
       | | cache the splitted PDU
     and then
       | operationResponse Op
```

5.2 Operações

Uma operação SNMP se realiza pela transmissão de um PDU, contendo o tipo da operação e os objetos que serão manipulados. Cada PDU é gerado em uma entidade e tratado em outra entidade. Os processos seguidos quando as operações de gerência são geradas e quando são tratadas serão vistos a seguir.

Primeiramente, serão abordadas as operações do protocolo propriamente ditas; em seguida, as operações realizadas sobre uma *VarBind* e, por último, as operações realizadas sobre os objeto

[3]Definido na Seção 5.4.

[4]Definida na Seção 5.4.

de gerência, que constituem a instrumentalização das MIBs.

5.2.1 Operações do protocolo

Esta seção trata da especificação das operações SNMP propriamente ditas, levando-se em consideração os passos realizados na sua geração (requisição das operações) e no seu tratamento (resposta das operações).

Trataremos as operações de requisição e de resposta. A operação *Report* não será abordada neste trabalho, pois, como dito em [CMRW96b], sua semântica não é definida pelo protocolo, sendo dependente da implementação.

Requisição de Operações

O processo de requisição de operações consiste em montar o PDU a ser enviado. Este processo será seguido para se gerar um PDU quando uma operação *Get*, *GetNext*, *GetBulk*, *Set*, *Inform* ou *Trap* tiver que ser enviada.

A ação operationRequest está relacionada com a composição de um PDU.

```
• operationRequest _ with (_, _, _) :: opTag, Index, Index, tuple → action
(14) operationRequest Op:opTag with (S:Index, I:Index, T:tuple) =
         | | generate request-id
         | and
         | | generate VarBindList from T
       then
         | give the PDU of (the given requestId#1, S, I, the given VarBindList#2) with tag Op
```

Inicialmente, um valor para *"request-id"* é gerado pela ação generate request-id[5]. Os valores para os campos *"non-repeaters"* e *"max-repetitions"*, usados quando a operação for *GetBulk*, são fornecidos pelas variáveis S e I. O último valor fornecido é uma tupla T, que será transformada numa *VarBindList* pela ação generate VarBindList from T, apresentada logo a seguir. Este é o único passo que varia em operationRequest, dependendo da operação a ser executada. Na última linha, um PDU é formado pelo construtor de PDUs the PDU of[6].

A ação generate VarBindList from _ possui três formas. Na primeira forma, uma *VarBindList* é formada a partir de uma lista de *ObjectName*:

```
• generate VarBindList from _ :: tuple → action
(15) generate VarBindList from N:list of ObjectName+ =
         | give N
       then
         | map using abstraction of emptyVarBind
```

[5]Definida na Seção 5.4.

[6]Definido na Seção 5.4.

A variável N é dada como informação transitória para a ação map using, que irá executar emptyVarBind[7] para cada *VarBind* na lista fornecida para formar *VarBinds* com valor nulo.

Na segunda forma, uma *VarBindList* é formada a partir de uma *VarBindList*. Como a lista já está pronta, só é necessário repassá-la como informação transitória. Este procedimento é necessário a fim de generalizar a ação operationRequest vista anteriormente.

```
(16) generate VarBindList from V:VarBindList = regive
```

Na terceira forma, uma *VarBindList* é gerada a partir de uma macro de notificação:

```
(17) generate VarBindList from M:Notification-Macro =
        | | | get("sysUpTime.0")
        | | then
        | | | give list of ("sysUpTime.0", the given ObjectValue)
        | and
        | | | get("snmpTrapOID.0")
        | | then
        | | | give list of ("snmpTrapOID.0", the given ObjectValue)
        | and
        | | | generate ObjectNameList from M
        | | then
        | | | map using abstraction of consultVarBind
        then
        | give concatenation(the given list#1, the given list#2, the given list#3)
```

Três listas são formadas e então concatenadas: uma contendo uma *VarBind* com o objeto *sysUpTime.0*, outra contendo uma *VarBind* com o objeto *snmpTrapOID.0*[8]. A terceira lista de *ObjectName* é gerada a partir da macro indicada por M, pela ação generate ObjectNameList from M, de forma dependente da implementação. Esta lista é então consultada na MIB local pela ação map using, que irá executar consultVarBind[9] para cada *VarBind* na lista fornecida. Na última linha, as três listas são concatenadas.

A ação generate ObjectNameList from M[10] gera uma lista contendo um ou mais *ObjectName* extraídos do campo OBJECTS de uma macro do tipo NOTIFICATION-TYPE, indicada por M.

Resposta de Operações

O processo de resposta de operações consiste em executar a operacão de gerência (consulta, alteração ou alerta), produzindo seu resultado. Este processo deve ser executado num ambiente que contenha as células de memória para salvar cada campo do PDU recebido (alocados em

[7] Definida na Seção 5.2.2.

[8] Estes dois valores são sempre inseridos no início de uma lista quando a operação for de notificação [CMRW96b, página 20].

[9] Definida na Seção 5.2.2.

[10] Definida na Seção 5.4.

initialize LCD-Primitive), retornando como informação transitória um *errorStatus* para *"error-status"*, um *Index* para *"error-index"* e uma *VarBindList* para *"variable-bindings"*. Se não houver erros, uma nova lista é gerada e os campos indicadores de erros são nulos. Em caso de um erro no processo de resposta, este erro e uma indicação para a respectiva variável em *"variable-bindings"* onde ele ocorreu são devolvidos com a lista originalmente fornecida.

A ação que executa tal procedimento é operationResponse, que difere para cada operação de gerência.

Em operationResponse Get (definida abaixo), somente a lista *"variable-bindings"* é fornecida como informação transitória para a execução de map using abstraction of consultVarBind. A ação map using executa consultVarBind[11] para cada variável contida em *"variable-bindings"*. Além disto, é retornado *noError* como *"error-status"* e 0 como *"error-index"* pela ação setError[12], para completar a tupla de saída da ação.

Em caso de erro em consultVarBind, é sinalizada uma exceção e a segunda subação do combinador trap é executada, onde é retornado *genErr* como *"error-status"* e o *Index* fornecido quando a ação escapou como *"error-index"*. A mesma *VarBindList* de origem será a de saída.

- operationResponse _ :: opTag → action

```
(18) operationResponse Get =
       | | setError(noError, 0)
       and
       | | | give the cached "variable-bindings"
       | | then
       | | | map using abstraction of consultVarBind
     trap
       | | setError(genErr, the given Index)
       and
       | | give the cached "variable-bindings"
```

A única diferença da ação operationResponse GetNext com relação à ação operationResponse Get é que a ação map using executa a ação consultNextVarBind ao invés de consultVarBind.

```
(19) operationResponse GetNext =
       | | setError(noError, 0)
       and
       | | | give the cached "variable-bindings"
       | | then
       | | | map using abstraction of consultNextVarBind
     trap
       | | setError(genErr, the given Index)
       and
       | | give the cached "variable-bindings"
```

A ação operationResponse GetBulk executa consultNextVarBind para cada variável contida na lista *"variable-bindings"*. Seja N o valor mínimo entre *"non-repeaters"* e o número de *VarBinds* contidas em *"variable-bindings"* do PDU recebido, e M o valor de *"max-repetitions"* do PDU

[11] Definida na Seção 5.2.2.

[12] Definida na Seção 5.4.

recebido[13]. Para as primeiras N variáveis contidas em *"variable-bindings"*, a consulta é efetuada apenas uma vez; para as demais variáveis, M vezes.

Inicialmente, N é dado, juntamente com a *VarBindList* original. O valor de N é usado para quebrar a lista em duas partes. Isto é feito pela ação break. Para a primeira parte da lista, consultNextVarBind será executada para cada *VarBind* pela ação map using. Para a segunda parte da lista, consultNextVarBind será executada M vezes para cada *VarBind*, sendo o valor de M fornecido para map with repetition using. As duas ações que processam listas devolvem, cada uma, uma lista com os valores consultados, que serão concatenadas a seguir. O procedimento de erro é o mesmo utilizado para a ação operationResponse Get.

```
(20) operationResponse GetBulk =
        | | setError(noError, 0)
        and
        | | | | give min(the cached "non-repeaters", count items of the cached "variable-bindings")
        | | | and
        | | | | give the cached "variable-bindings"
        | | then
        | | | break(the given list#2, the given integer#1)
        | | then
        | | | | | give the given list#1
        | | | | then
        | | | | | map using abstraction of consultNextVarBind
        | | | and then
        | | | | | | give the cached "max-repetitions"
        | | | | | and
        | | | | | | give the given list#2
        | | | | then
        | | | | | map with repetition using abstraction of consultNextVarBind
        | | | then
        | | | | give concatenation(the given list#1, the given list#2)
        | | trap
        | | | setError(genErr, the given Index)
        | | and
        | | | give the cached "variable-bindings"
```

A ação operationResponse Set é executada em dois passos distintos: inicialmente a *VarBindList* é passada para a ação map using, que executa validateVarBind para cada variável contida na lista salva como *"variable-bindings"*, a fim de testar se todas as atribuições podem ser efetuadas. A ação validateVarBind verifica diversas condições, inclúindo se a variável existe, sua forma de acesso e outras mais.

Se tudo suceder sem erros, então numa segunda fase todas as alterações são realizadas pela ação updateVarBind e é devolvido *"error-status"* setado como *noError*, *"error-index"* igual a 0 e a *VarBindList* original associada a *"variable-bindings"*. Se algum erro acontecer nesta segunda fase, todas as alterações devem ser desfeitas (mapeando a lista usando a ação undoVarBind) e é devolvido *"error-status"* setado como *commitFailed*, *"error-index"* igual ao índice da variável onde o erro ocorreu e a *VarBindList* original associada a *"variable-bindings"*. Então, caso não

[13]Os valores de M e N são definidos em [CMRW96b].

seja possível desfazer alguma das alterações já efetuadas, é devolvido *"error-status"* setado como *undoFailed*, *"error-index"* igual a 0 e a *VarBindList* original associada a *"variable-bindings"*.

```
(21) operationResponse Set =
        | | | give the cached "variable-bindings"
        | | then
        | | | map using abstraction of validateVarBind
        | | then
        | | | | setError(noError, 0)
        | | | and
        | | | | map using abstraction of updateVarBind
        | | trap
        | | | | setError(commitFailed, the given Index)
        | | | and
        | | | | | break(the cached "variable-bindings", the given Index)
        | | | | then
        | | | | | map using abstraction of undoVarBind
        | | trap
        | | | | setError(undoFailed, 0)
        | | trap
        | | | setError(the given errorStatus#1, the given Index#2)
        | then
        | | give (the given errorStatus#1, the given Index#2, the cached "variable-bindings")
```

Na ação operationResponse Inform, retorna-se *"error-status"*, *"error-index"* e *"variable-bindings"* como informação transitória para que seja enviada de volta para a entidade que gerou esta operação. Então, uma aplicação é selecionada pelo produtor the Application associated with[14], dependendo do índice do atual tipo de notificação, dado pelo produtor NotificationTypeId of[15]. Em seguida, a *VarBindList* recebida é enviada para esta aplicação, que executará uma ação dependente da implementação que realize a notificação pedida.

```
(22) operationResponse Inform =
        | setError(noError, 0)
        and
        | give the cached "variable-bindings"
        and
        | | give the Application associated with NotificationTypeId of the cached "variable-bindings"
        | then
        | | send a message [to the given Application][containing the cached "variable-bindings"]
```

A ação operationResponse Trap executa exatamente os mesmos passos que operationResponse Inform, sem retornar informação transitória, pois a operação *Trap* não necessita de resposta.

```
(23) operationResponse Trap =
        | give the Application associated with NotificationTypeId of the cached "variable-bindings"
        then
        | send a message [to the given Application][containing the cached "variable-bindings"]
```

Quando a aplicação é identificada, o processamento das operações *Inform* e *Trap* passa a ser dependente da implementação, e não precisa ser abordado aqui, pois para cada notificação diferente, há uma ação distinta.

[14]Definido na Seção 5.4.

[15]Definido na Seção 5.4.

5.2.2 Operações sobre *VarBinds*

Esta seção define as ações que são executadas quando uma variável do sorte *VarBind* é processada.

A ação emptyVarBind possui a função de limpar o valor de uma variável do sorte *VarBind*. Para tanto, dado um *VarBind*, ou apenas um *ObjectName*, ela gera um *VarBind* mantendo o nome do objeto fornecido e estabelecendo *unSpecified* como seu valor.

- emptyVarBind :: action[receiving VarBind][giving VarBind]

```
(24) emptyVarBind =
         give (the given ObjectName#1, unSpecified)
```

A ação consultVarBind consulta o valor de uma variável do sorte *VarBind*, fornecida como informação transitória numa MIB local. Para tanto, os seguintes testes são efetuados: o produtor existsObject[16] pega o nome do objeto de gerência contido na *VarBind* recebida e reconhece se este nome indica um objeto válido ou não. Caso o teste não proceda, é retornada uma *VarBind* contendo o *ObjectName* fornecido e o valor de erro *noSuchObject*. Caso o teste proceda, será verificado se o objeto está presente ou não nas MIBs da entidade atual pelo produtor existsInstance[17]. Caso o teste não proceda, é retornada uma *VarBind* contendo o *ObjectName* fornecido e o valor de erro *noSuchInstance*. Caso esta avaliação proceda, então é retornada uma tupla contendo o *ObjectName* fornecido e seu valor consultado pela ação get[18].

Se por qualquer motivo a ação get não conseguir finalizar a consulta, é sinalizada uma excessão para fazer a ação consultVarBind terminar excepcionalmente.

- consultVarBind :: action[receiving VarBind][giving VarBind]

```
(25) consultVarBind =
         | | check (existsObject the given ObjectName#1)
         and then
         | | | | check not (existsInstance the given ObjectName#1)
           and then
           | | give the given ObjectName#1
           and
           | | | get(the given ObjectName#1)
         else
         | | give (the given ObjectName#1, noSuchInstance)
       else
       | give (the given ObjectName#1, noSuchObject)
```

A ação consultNextVarBind consulta o valor da próxima variável com nome lexicograficamente superior a uma variável do sorte *VarBind* fornecida como informação transitória, numa MIB local. Para tanto, o nome do próximo objeto lexicograficamente superior ao *ObjectName*

[16] Definido na Seção 5.4.

[17] Definido na Seção 5.4.

[18] Definido na Seção 5.3.

fornecido que exista em uma MIB local é obtido pela ação the next to[19] e então é retornada uma tupla contendo o nome deste objeto e o seu valor consultado pela ação get, caso a avaliação proceda. Caso não exista nenhum objeto lexicograficamente superior ao *ObjectName* fornecido é retornado uma tupla contendo o *ObjectName* original e o valor de erro *endOfMibView*.

- consultNextVarBind :: action[receiving VarBind][giving VarBind]

```
(26) consultNextVarBind =
         | | give the next to the given ObjectName#1
         | then
         | | | regive
         | | and
         | | | get(the given ObjectName)
         | else
         | give (the given ObjectName#1, endOfMibView)
```

A ação validateVarBind analisa diversas condições que devem ser validadas antes que uma alteração seja efetuada num objeto de uma MIB. Os seguintes testes são efetuados: (1) o nome da variável é validado; (2) o tipo do valor a ser atribuído deve ser o mesmo aceito pela variável na MIB; (3) o tamanho do valor a ser atribuído deve ser o mesmo aceito pela variável na MIB; (4) as regras de codificação do valor a ser atribuído devem ser as mesmas aceitas pela variável na MIB; (5) o valor a ser atribuído deve ser um valor que pertença ao domínio do tipo da variável na MIB; (6) o nome da variável fornecido deve ser um nome que pode ser criado; (7) qualquer condição que impeça a criação da variável neste instante deve ser testada então; (8) o tipo de acesso da variável deve permitir escrita, ou seja, não deve ser Not-Accessible ou Read-Only; (9) qualquer condição que impeça a atribuição neste momento deve ser testada então. A ação que testa estas condições é ifnot Y generate error E[20], que avalia um produtor Y e caso este não proceda, o valor E é devolvido como índice de um erro.

- validateVarBind :: action[receiving VarBind]

```
(27) validateVarBind =
```

[19] Definido na Seção 5.4.

[20] Definida na Seção 5.4.

```
| ifnot (existsObject the given ObjectName#1)
|       generate error notWritable
and then
| ifnot (type(the given VarBind) is type(the given ObjectName#1))
|       generate error wrongType
and then
| ifnot (length(the given VarBind) is length(the given ObjectName#1))
|       generate error wrongLength
and then
| ifnot (encoding(the given VarBind) is encoding(the given ObjectName#1))
|       generate error wrongEncoding
and then
| ifnot (alwaysAtrib(the given ObjectName#1, the given ObjectSyntax#2))
|       generate error wrongValue
and then
| ifnot (alwaysCreate(the given ObjectName#1))
|       generate error noCreation
and then
| ifnot (nowCreate(the given ObjectName#1))
|       generate error inconsistentName
and then
| ifnot (not access(the given ObjectName#1) is READ-ONLY and
|       not access(the given ObjectName#1) is NOT-ACCESSIBLE)
|       generate error notWritable
and then
| ifnot (nowAtrib(the given ObjectName#1, the given ObjectSyntax#2))
|       generate error inconsistentValue
```

A ação updateVarBind altera o valor de um objeto numa MIB. Esta função só deve ser utilizada após a ação validateVarBind ter sido executada.

- updateVarBind :: action[receiving VarBind][giving VarBind]

```
(28) updateVarBind =
         set(the given ObjectName#1, the given ObjectSyntax#2)
```

A ação undoVarBind desfaz a última alteração feita em um objeto numa MIB. Esta ação é dependente da implementação.

- undoVarBind :: action

```
(29) undoVarBind = □
```

Para que uma destas ações seja repetida para uma *VarBindList*, basta utilizar a ação map using [21], fornecendo a abstração da ação a ser executada como parâmetro a ela.

5.3 Operações sobre Informações de Gerência

Esta seção define as ações necessárias para controlar e monitorar os objetos contidos em MIBs. Para cada objeto que exista em MIBs, deve ser construída uma instância destas ações e produtores. Especificar esta notação para todos os objetos de gerência já existentes está fora do contexto deste trabalho, cabendo aqui apenas definir de forma genérica sua nomenclatura e utilização. Por isto, neste ponto da especificação, estas ações serão dependentes da implementação.

[21] Definido na Seção 5.4.

A ação get(N) retorna o valor associado ao objeto de nome N numa MIB local como informação transitória.

- get(_) :: ObjectName → action[giving ObjectValue]

(30) get(N:ObjectName) = □

A ação set(N, V) altera o valor do objeto N em uma MIB local para o valor V. A ação recebe um *ObjectName* e um *ObjectSyntax* como parâmetros. Os mesmos valores são retornados como informação transitória.

- set(_,_) :: ObjectName, ObjectSyntax → action

(31) set(N:ObjectName,S:ObjectSyntax) = □

O produtor existsObject N retorna verdadeiro caso o nome do objeto N fornecido seja um nome de objeto válido.

- existsObject _ :: ObjectName → yielder[of truth-value]

(32) existsObject N:ObjectName = □

O produtor existsInstance N retorna verdadeiro caso o objeto N exista em alguma MIB na entidade local. Este produtor deve ser diferente em cada implementação, pois somente variáveis que existam na MIB consultada devem retornar um valor verdadeiro.

- existsInstance _ :: ObjectName → yielder[of truth-value]

(33) existsInstance N:ObjectName = □

O produtor the next to N devolve o nome da variável lexicograficamente superior ao nome da variável N fornecida que exista numa MIB local. O produtor falha caso não exista nenhuma variável nesta condição.

- the next to _ :: ObjectName → yielder[of ObjectName]

(34) the next to N:ObjectName = □

Os seguintes produtores testam variáveis do sorte *VarBind* e variáveis em MIB e são dependentes da implementação:

- access(_): retorna o tipo de acesso que um objeto possui, que é um valor do sorte *MaxAccess*.
- type(_): retorna o tipo de um *VarBind* ou de um objeto de uma MIB.
- length(_): retorna o tamanho de um *VarBind* ou de um objeto de uma MIB. Este teste é importante para se saber se um valor a ser atribuído possui mais *bits* que a memória que vai contê-lo.
- encoding(_): retorna o *Basic Encoding Rules* (BER) utilizado por um *VarBind* ou por um objeto de uma MIB.
- alwaysCreate(_): retorna verdadeiro caso um objeto possa sempre ser criado; retorna falso caso nunca possa ser criado.

- nowCreate(_): retorna verdadeiro caso um objeto possa ser criado neste instante; retorna falso caso um objeto possa ser criado, mas não neste instante.
- alwaysAtrib(_): retorna verdadeiro caso um objeto possa sempre ser atribuído; retorna falso caso nunca possa ser atribuído.
- nowAtrib(_): retorna verdadeiro caso um objeto possa ser atribuído neste instante; retorna falso caso um objeto possa ser atribuído, mas não neste instante.

(35) access(N:ObjectName) = □
(36) type(V:VarBind) = □
(37) type(N:ObjectName) = □
(38) length(V:VarBind) = □
(39) length(N:ObjectName) = □
(40) encoding(V:VarBind) = □
(41) encoding(N:ObjectName) = □
(42) alwaysCreate(N:ObjectName) = □
(43) nowCreate(N:ObjectName) = □
(44) alwaysAtrib(N:ObjectName, S:ObjectSyntax) = □
(45) nowAtrib(N:ObjectName, S:ObjectSyntax) = □

5.4 Notação Auxiliar

Esta seção define dados, ações e produtores auxiliares utilizados na especificação das operações SNMP, incluindo abreviaturas de entidades semânticas.

5.4.1 Ações

As ações map using e map with repetition using são usadas para processar qualquer tipo de lista, sendo neste trabalho utilizadas somente para lidar com *VarBindList*.

Antes de mostrar suas especificações, pretende-se explicar como é o mecanismo geral de uma ação que lida com listas. Uma lista pode ser particionada em cabeça e cauda pelas funções head e tail, respectivamente. Desta forma, pode-se proceder com alguma ação sobre um único item da lista quando este é isolado pela função head. O processo pode ser repetido recursivamente para toda lista através da ação unfold passando somente sua cauda, até que a lista esteja vazia.

A ação map using recebe uma *VarBindList* como informação transitória e executa a ação A indicada para cada *VarBind* da lista. Ela é análoga à função **map** em linguagens funcionais.

- map using _ :: abstraction → action[receiving list]

```
(46) map using A:abstraction =
       unfolding
       | | | check ( the given list is empty-list )
       | | and then
       | | | give it
       | or
       | | | check not ( the given list is empty-list )
       | | and then
       | | | | | give head of the given list
       | | | | then
       | | | | | enact A
       | | | | then
       | | | | | give list of the given tuple
       | | | and
       | | | | | give tail of the given list
       | | | | then unfold
       | | then
       | | | | give concatenation(the given list#1, the given list#2)
```

A ação map with repetition using recebe um inteiro M e uma *VarBindList* como informação transitória e faz equivalente a map using, mas repete todo o processo M vezes.

- map with repetition using _ :: abstraction → action[receiving (natural, list)]

```
(47) map with repetition using A:abstraction =
       unfolding
       | | | check ( the given natural#1 is 0 )
       | | and then
       | | | give the empty-list
       | or
       | | | check not ( the given natural#1 is greater than 0 )
       | | and then
       | | | | | give difference(the given natural#1, 1)
       | | | | and
       | | | | | | give the given list#2
       | | | | then
       | | | | | | map using A
       | | | then
       | | | | | give the given list#2
       | | | | and
       | | | | | unfold
       | | | then
       | | | | | give concatenation(the given list#1, the given list#2)
```

A ação break(L, I) quebra uma lista L qualquer em duas listas, deixando os I primeiros itens da lista original na primeira lista e os itens restantes na segunda lista. Caso haja menos de I itens na lista original, todos os itens ficarão na primeira lista e a segunda será uma lista vazia.

- break _ :: (list, integer) → action[giving (list, list)]

```
(48) break (L:list, I:integer) =
```

```
| give 0 and give L
then
| unfolding
| | | ckeck (the given list#2 is empty-list)
| | and then
| | | give (empty-list, empty-list)
| | or
| | | ckeck (the given integer#1 is I)
| | and then
| | | give (empty-list, the given list#2)
| | or
| | | ckeck (the given integer#1 is less than I)
| | and then
| | | | give list of head of the given list#2
| | | and
| | | | | give sum(the given integer#1, 1)
| | | | and
| | | | | give tail of the given list#2
| | | then
| | | | unfold
| | then
| | | give (concatenation(the given list#1, the given list#2), the given list#3)
```

A ação accept M recebe uma mensagem M, e então o remetente da mensagem é salvo com o rótulo *"sender"*, e o conteúdo da mensagem é retornado como informação transitória.

- accept _ :: message → action[giving tuple]

```
(49) accept M:message =
        | receive M
        then
        | | cache the sender of it as "sender"
        | and
        | | give the contents of it
```

A ação accept contents of M é uma forma reduzida da ação accept M que retorna apenas o conteúdo da mensagem recebida como informação transitória, sem guardar a identidade do seu emissor. É utilizada quando não é necessário devolver uma resposta.

- accept contents of _ :: message → action[giving tuple]

```
(50) accept contents of M:message =
        | receive M
        then
        | give the contents of it
```

A ação ifnot Y generate error E testa um produtor Y, que deve retornar um valor transitório do sorte *truth-value*. Se este produtor resultar em *nothing*, a ação retorna uma tupla contendo o erro E e o índice da atual *VarBind* processada em *"variable-bindings"*, dado pelo produtor index of N in V. Caso contrário, a ação termina normalmente.

- ifnot _ generate error _ :: yielder[of truth-value], errorStatus → action[receiving ObjectName]

```
(51) ifnot Y:yielder generate error E:errorStatus =
        | check Y
        or
        | | check not Y
        | and then
        | | escape with (E, index of the given ObjectName in the cached "variable-bindings")
```

A ação cache D as K salva o valor D do sorte *data* associado ao identificador K. Este valor guardado pode ser acessado através do produtor the cached K.

- cache _ as _ :: tuple, token → action

```
(52) cache D:tuple as K:token =
         store D in the cell bound to K
```

A ação cache the splitted PDU recebe como informação transitória valores referentes aos campos de um PDU, que são *"request-id"*, *"error-status"*, *"error-index"* e *"variable-bindings"*, e salva estes campos nas células com rótulos já criados na ação initialize LCD-primitive.

Dois tipos de PDUs são definidos em [CMRW96b]: *StandardPDU* e *BulkPDU*. Ambos são idênticos quanto ao número, ordem e tipo de seus campos, existindo diferenças apenas em sua nomenclatura[22]. Como a ação cache the splitted PDU somente é utilizada na resposta de requisições, e não em sua solicitação, e os campos *"error-status"* e *"error-index"* só são utilizados a partir do momento em que uma resposta é gerada, a ação identifica estes campos sempre como *"non-repeaters"* e *"max-repetitions"*. Além disto, se o valor de *"non-repeaters"* for menor que 0, este valor é abandonado e 0 é associado em seu lugar, como mencionado em [CMRW96b, Seção 4.2.3].

```
• cache the splitted PDU :: action
(53) cache the splitted PDU =
         | cache the given requestId#1 as "request-id"
         and
         | cache max(0, the given Index#2) as "non-repeaters"
         and
         | cache the given Index#3 as "max-repetitions"
         and
         | cache the given VarBindList#4 as "variable-bindings"
```

A ação setError(S,I) devolve uma tupla contendo os valores relativos a *"error-status"* e *"error-index"*, fornecidos como S e I.

```
• setError _ :: (errorStatus, Index) → action[giving (errorStatus, Index)]
(54) setError(S:errorStatus, I:Index) =
         | give (S, I)
```

A ação generate request-id gera rótulos para indexar os PDUs que fluem entre as entidades. Ela devolve como informação transitória um valor para *"request-id"*, que irá integrar um PDU para identificá-lo. Esta ação deve identificar os pacotes em tramitação para que não sejam fornecidos códigos idênticos a PDUs distintos. A ação generate request-id é dependente da implementação.

```
• generate request-id :: action[giving requestId]
(55) generate request-id = □
```

A ação generate ObjectNameList from M gera uma lista contendo diversos *ObjectName* que possam ser extraídos a partir do processamento do campo OBJECTS de uma macro do tipo

[22] Os campos denominados de *"error-status"* e *"error-index"* no *StandardPDU* são denominados de *"non-repeaters"* e *"max-repetitions"* no *BulkPDU*; eles são usados com semânticas diferentes, apesar dos tipos serem os mesmos.

NOTIFICATION-TYPE indicada por M (este processamento deve ser uma extração sintática dos nomes dos objetos). Como dito anteriormente, esta ação é dependente da implementação.

- generate ObjectNameList from _ :: Notification-Macro → action[giving ObjectNameList]

(56) generate ObjectNameList from M:Notification-Macro = □

5.4.2 Produtores

O produtor the cached K recupera o valor guardado com o rótulo K, pela ação cache.

- the cached _ :: token → yielder[of datum]

(57) the cached K:token =
the datum stored in the cell bound to K

O produtor the A associated with D retorna o agente da semântica de ações de tipo A (aplicação, despachante ou outro) associado com a tupla D de informação.

A forma pela qual uma aplicação é escolhida para processar a resposta de uma notificação não é nada clara na especificação informal contida em [CMRW96b]. Por isto, convencionou-se neste trabalho fornecer uma constante (*Notification-Macro*) como informação para a escolha da aplicação, ao invés de uma variável, já que o valor a ser passado para a escolha não é definido na especificação informal. O produtor the Application associated with Notification-Macro representa esta escolha, sendo dependente da implementação.

- the _ associated with _ :: agent, tuple → yielder[of agent]

(58) the $A \leq$ Application associated with M:Notification-Macro = □

O produtor index of N in V determina o índice da variável com nome N na *VarBindList* V.

- index of _ in _ :: ObjectName, VarBindList → yielder[of Index]

(59) index of N:ObjectName in V:VarBindList = □

O produtor NotificationTypeId of V determina o identificador da notificação que está sendo veiculada na *VarBindList* V. Para isto, o nome da segunda *VarBind* da lista é consultado. Se este nome for *"snmpTrapOID.0"* então seu valor é retornado. Caso contrário, a *VarBindList* não contém uma notificação, caso em que o produtor retorna *nothing*.

- NotificationTypeId of _ :: VarBindList → yielder[of ObjectValue]

(60) NotificationTypeId of V:VarBindList = □

5.4.3 Dados

A definição formal dos sortes relacionados com os objetos de gerência e as operações de gerência é muito simples, consistindo na tradução de sua especificação em SMI, que se encontra em [CMRW96b, Seção 3], para a notação de dados usada em semântica de ações.

Os sortes definidos como SEQUENCE em SMI são traduzidos como tuplas contendo todos os campos da sequência. Os sortes definidos como CHOICE em SMI são traduzidos como uniões de sortes. A notação □ significa que o lado esquerdo da equação não é completamente definido neste estágio da especificação, podendo vir a conter outros sortes, dependendo de cada caso na implementação.

As seguintes definições são derivadas da especificação informal em [CMRW96b]. Os elementos do sorte *errorStatus* são indicações de erro; sua explicação se encontra durante sua ocorrência no texto ou na especificação informal. *errorStatus* é um subsorte de *Index*, que por sua vez é um subsorte de *integer*. *VarBind* representa uma instância de um objeto de gerência, uma variável de gerência, contendo seu nome (*ObjectName*) e valor (*ObjectValue*), que pode ser um valor dependente da implementação (*objectSyntax*) ou uma indicação de erro (*Exceptions*). *VarBindList* é uma lista de VarBind.

(61) handle $\geq$ requestId

(62) errorStatus = noError | genErr | commitFailed | undoFailed | wrongType | wrongLength | wrongEncoding | wrongValue | noCreation | inconsistentValue | notWritable | inconsistentName | □ (*individual*)

(63) errorStatus $\leq$ Index

(64) Index $\leq$ integer

(65) VarBindList = list of VarBind^{+}

(66) VarBind = (ObjectName, ObjectValue)

(67) ObjectName $\leq$ token

(68) ObjectValue = ObjectSyntax | Exceptions

(69) ObjectSyntax = □

(70) Exceptions = unSpecified | noSuchObject | noSuchInstance | endOfMibView (*individual*)

As seguintes definições são referentes a formação e destruição de PDUs. O sorte PDU define o pacote que é trocado entre as entidades. O sorte *opTag* define rótulos que irão marcar os PDUs como sendo de uma determinada operação. O axioma 72 é um contrutor de PDUs. Os axiomas de 73 a 77 são destrutores de PDUs. O axioma 78 retorna um PDU quando este for rotulado pela operação solicitada.

(71) opTag = Get | GetNext | GetBulk | Set | Inform | Trap | Response | □ (*individual*)

(72) PDU of (R:requestId, S:errorStatus, I:Index, V:VarBindList) with tag T:tag $\rightarrow$ PDU

(73) request-id of PDU of (R,S,I,V) with tag $T = R$

(74) error-status of PDU of (R,S,I,V) with tag $T = S$

(75) error-index of PDU of (R,S,I,V) with tag $T = I$

(76) variable-bindings of PDU of (R,S,I,V) with tag $T = V$

(77) operation of PDU of (R,S,I,V) with tag $T = T$

- _ tagged with _ :: PDU, tag $\rightarrow$ PDU

(78) P:PDU tagged with Op:opTag =
if operation of P is Op then P else nothing

As seguintes definições são utilizadas pela notação auxiliar descrita na seção anterior. Até aqui, a definição de agente da semântica de ações considera que uma entidade é composta por uma única aplicação. O reconhecimento de uma entidade é um problema de rede (uma entidade falha ou um erro no seu roteamento) e pertence a outro estágio da especificação.

(79) agent = entity | Application | □

(80) entity = □

(81) Notification-Macro = □

(82) MaxAccess = Not-Acessible | Read-Only | Read-Create | Read-Write | □

5.5 Considerações Finais

Este capítulo tratou de definição formal das operações SNMP. Estas operações serão usadas pelas aplicações padrão SNMP (Capítulo 7) tanto para gerar quanto para responder a requisições. Isto inclui as ações operationRequest e operationResponse. A notação auxiliar aqui definida será reutilizada sem ser redefinida nos capítulos posteriores. Somente novas ações serão explicadas.

CAPÍTULO 6

Despachante SNMP

Neste capítulo, o Despachante SNMP é especificado. A Seção 6.1 define a especificação de alto nível de um Despachante. A Seção 6.2 aborda o processo seguido pelo Despachante para enviar uma requisição, a Seção 6.3 aborda o seu processo para enviar uma resposta, enquanto que a Seção 6.4 aborda o seu processo para receber requisições e respostas. A notação auxiliar definida neste capítulo é apresentada na Seção 6.5.

Este capítulo inclui e complementa o trabalho apresentado em [FMD00b]. Toda a especificação a seguir é efetuada a partir da especificação informal do Despachante apresentada em [CHPW99].

A partir deste capítulo, vai se considerar a entidade definida em [HPW99] e mostrada informalmente no Capítulo 3. Esta entidade utiliza as operações definidas no Capítulo 5 para trocar dados com outras entidades. Este capítulo define apenas o procedimento seguido pelo módulo Despachante. As aplicações padrão são definidas no próximo capítulo e no Capítulo 8 uma entidade é montada com todos seus módulos.

6.1 Procedimento Despachante

O Despachante é o módulo do engenho responsável pelo controle do fluxo de mensagens entre os subsistemas de uma entidade, pelo controle do fluxo de mensagens com outras entidades na rede e pela escolha dos serviços providos para estas mensagens.

Para que isto ocorra, quatro tipos de procedimentos são seguidos por um Despachante:

1. envio de uma requisição, quando uma aplicação (Gerador de Comandos or Gerador de Notificações) requer que uma mensagem seja enviada pela rede;
2. recepção de uma requisição, quando uma mensagem recebida é processada e enviada para uma aplicação (Receptor de Comandos ou Receptor de Notificações);
3. envio de uma resposta, quando uma aplicação (Receptor de Comandos ou Receptor de Notificações) requer que o despachante envie uma resposta para uma requisição previamente

recebida;

4. recepção de uma resposta, quando uma mensagem contendo uma resposta é retornada para a aplicação (Gerador de Comandos or Gerador de Notificações) que enviou uma requisição previamente feita.

A especificação em semântica de ações do Despachante é escrita de forma *top-down*. A seguinte ação representa sua estrutura geral:

```
• procedure Dispatcher :: action
(1) procedure Dispatcher =
        | accept a message[from an agent][containing a SDU]
      then
        | | procedure Disp send-request
        or
        | | procedure Disp send-response
        or
        | | procedure Disp receive
```

O Despachante pode receber mensagens de agentes da semântica de ações atuando como modelos de processamento de mensagens ou aplicações (dentro da entidade SNMP ao qual o Despachante pertence), tão bem quanto de agentes da semântica de ações atuando como Despachantes pertencentes a outras entidades.

Inicialmente, o Despachante recebe uma mensagem provinda de uma aplicação local ou de outras entidades. Então, uma ação é escolhida para ser executada, dependendo do conteúdo da mensagem recebida: a ação procedure Disp send-request é executada quando um *sendPdu-SDU* é recebido de uma aplicação local, para enviar uma requisição para outras entidades. A ação procedure Disp send-response é executada quando um *returnResponsePdu-SDU* é recebido de uma aplicação local, para enviar uma resposta para outras entidades. e a ação procedure Disp receive é executada quando um *Network-MSG* é recebido de outras entidades. Este caso corresponde à recepção de requisições ou respostas, as quais serão passadas para uma aplicação dentro da entidade. A ação accept já foi vista na Seção 5.4.

De acordo com a especificação informal do Despachante [CHPW99, chapter 4], mensagens de erro cessam o processamento corrente. Esta característica é modelada através de terminação por exceção (ação escape) dentro das subações do Despachante, cada vez que uma mensagem de erro for identificada.

A ação procedure Disp send-request é dada na próxima seção. As ações procedure Disp send-response e procedure Disp receive são análogas à ação procedure Disp send-request e são mostradas logo a seguir.

6.2 Procedimento Send-Request

Este procedimento é seguido dentro de um despachante quando uma aplicação (Gerador de Comandos ou Gerador de Notificações) deseja que uma requisição seja enviada a outra entidade.

A ação procedure Disp send-request é definida como uma sequência, na qual:

1. O sorte dos dados recebidos é checado. Note que a ação give the given X falha quando a informação transitória fornecida a ela não for do sorte X; neste caso, deve ser um *sendPdu-SDU*, provindo de uma aplicação.

2. Uma comunicação com o agente da semântica de ações executando um modelo de processamento de mensagens dentro da própria entidade é estabelecida, baseado no valor do campo *messageProcessingModel* provido pela aplicação no *sendPdu-SDU* recebido.

3. Uma mensagem é montada e enviada para uma entidade externa.

```
• procedure Disp send-request :: action[receiving tuple]
(2) procedure Disp send-request =
        | | give (the given tuple)[sendPdu-SDU]
        | then
        | | procedure Disp send-request A
        | then
        | | send a message[to the MPModel associated with the given messageProcessingModel#3]
        | |                [containing (the given tuple)[prepareOutgoingMsgIn-SDU]]
      and then
        | | accept contents of a message[from an MPModel][containing a tuple[prepareOutgoingMsgOut-SDU]]
        | then
        | | procedure Disp send-request B
        | then
        | | send a message[to the Dispatcher associated with (the given transportDomain#1,
        | |                the given transportAddress#2)][containing the given Network-MSG#3]
```

O produtor the MPModel associated with[1] resulta num agente da semântica de ações atuando como modelo de processamento de mensagens dentro da entidade e the Dispatcher associated with[2] resulta num agente da semântica de ações atuando como Despachante de uma entidade externa. Suas definições são específicas para cada entidade SNMP, e irão depender de parâmetros da especificação.

Os dados trocados entre os módulos da entidade quando uma requisição deve ser enviada são mostrados na Figura 6.1. Sequencialmente, acontecem as seguintes transmissões: uma aplicação envia um *sendPdu-SDU* para o Despachante; então o Despachante envia um *prepareOutgoingMsgIn-SDU* para um modelo de processamento de mensagem e recebe um *prepareOutgoingMsgOut-SDU* deste mesmo modelo. Por último, o Despachante envia um *Network-MSG* para a rede.

[1] Definido na Seção 6.5.

[2] Definido na Seção 6.5.

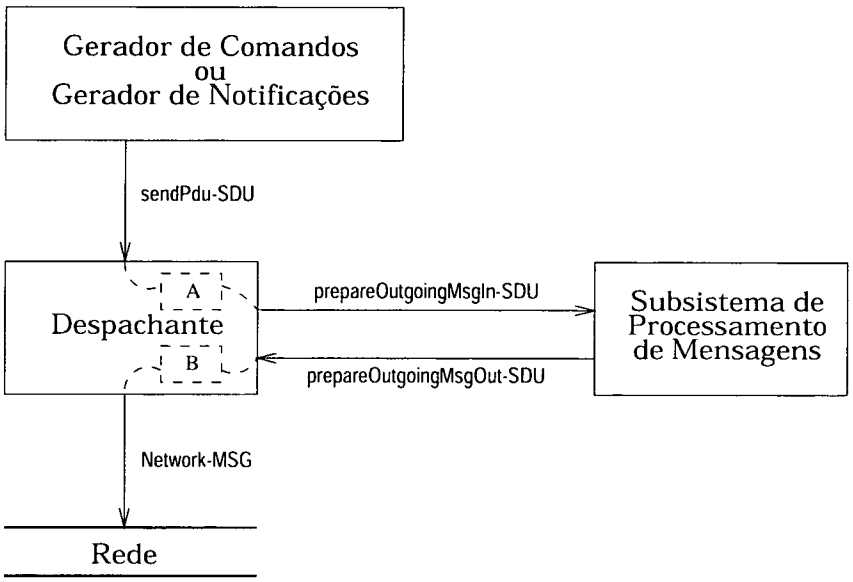

Figura 6.1: Procedimento de envio de uma requisição e troca dos SDUs.

A seguir serão definidas as ações procedure Disp send-request A e procedure Disp send-request B. Ambas possuem dois estágios: a *análise* dos itens de dados recebidos e a *tradução* destes itens em um novo pacote, o qual será transmitido como conteúdo de uma mensagem. A ação procedure Disp send-request A corresponde ao estágio A dentro do Despachante (Figura 6.1):

- procedure Disp send-request A :: action[receiving sendPdu-SDU][giving prepareOutgoingMsgIn-SDU]

```
(3)  procedure Disp send-request A =
         | | check (the MPModel associated with the given messageProcessingModel#3 is an agent)
         or
         | | | check not (the MPModel associated with the given messageProcessingModel#3 is an agent)
         | | and then
         | | | send noMPModel back to sender
         | | and then
         | | | escape
       and then
       | regive and generate SendPduHandle
```

No estágio de análise (do topo da ação até o and then principal), o modelo de processamento de mensagem é validado, checando se o produtor the MPModel associated with _ retorna um agente da semântica de ações. A ação send _ back to sender[3], envia o índice de erro noMPModel de volta à aplicação remetente quando a comparação executada pela ação check resultar em *false*.

No caso em que o modelo de processamento de mensagens é válido, a ação passa para o estágio de tradução (a última linha da ação). Neste estágio, todos os dados recebidos (*sendPdu-SDU*) são propagados e o índice *sendPduHandle* é gerado pela ação generate sendPduHandle[4],

[3]Definida na Seção 6.5.

[4]Definido na Seção 6.5.

para identificar a requisição, e é então agregado à tupla, para formar uma tupla do tipo *prepareOutgoingMsgIn-SDU*.

A seguir será definida a ação procedure Disp send-request B, que é responsável por preparar requisições que serão enviadas para o Despachante de uma entidade externa. Ela corresponde ao estágio B dentro do Despachante (Figura 6.1):

- procedure Disp send-request B :: action[receiving prepareOutgoingMsgOut-SDU]

```
(4)  procedure Disp send-request B =
         | | | check (the given statusInformation#1 is success)
         | | and then
         | | | send the generated sendPduHandle back to sender
         | or
         | | | check not (the given statusInformation#1 is success)
         | | and then
         | | | send the given statusInformation#1 back to sender
         | | and then
         | | | escape
         and then
         | give (the given transportDomain#2, the given transportAddress#3, the given Network-MSG#4)
```

No estágio de análise (do topo da ação até o and then principal), *statusInformation* é comparado com *success*. Sucesso significa que nenhum erro ocorreu no procedimento seguido pelo modelo de processamento de mensagens.

No caso em que *statusInformation* é *success*, a ação passa para o estágio de tradução (a última linha da ação). Um *prepareOutgoingMsgOut-SDU* é recebido como informação transitória e então o *transportData* e o *Network-MSG* recebidos são propagados, sendo que o *Network-MSG* será enviado para outra entidade.

6.3 Procedimento Send-Response

Este procedimento é executado dentro de um despachante quando uma aplicação (Receptor de Comandos ou Receptor de Notificações) requer que uma resposta seja enviada a outra entidade.

A ação procedure Disp send-response é definida como uma seqência, na qual:

1. O sorte dos dados recebidos é checado, e deve ser um *returnResponsePdu-SDU*, provindo de uma das aplicações acima citada.
2. Uma comunicação com o agente da semântica de ações executando um modelo de processamento de mensagens dentro da própria entidade é estabelecida, baseado no valor do campo *messageProcessingModel* provido pela aplicação no *returnResponsePdu-SDU* recebido.
3. Uma mensagem é montada e enviada para uma entidade externa.

- procedure Disp send-response :: action[receiving tuple]

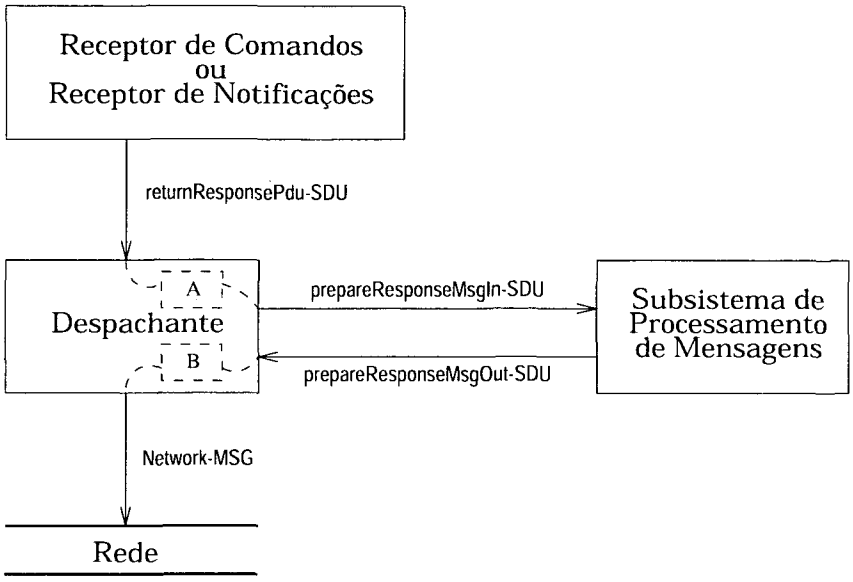

Figura 6.2: Procedimento de envio de uma resposta e troca dos SDUs.

```
(5)  procedure Disp send-response =
         | | give (the given tuple)[returnResponsePdu-SDU]
       then
         | | send a message[to the MPModel associated with the given messageProcessingModel#1]
                          [containing (the given tuple)[prepareResponseMsgIn-SDU]]
     and then
         | | accept contents of a message[from an MPModel][containing a tuple[prepareResponseMsgOut-SDU]]
       then
         | procedure Disp send-response B
       then
         | | send a message[to the Dispatcher associated with (the given transportDomain#1,
                          the given transportAddress#2)][containing the given Network-MSG#3]
```

Os dados trocados entre os módulos da entidade quando uma resposta deve ser enviada são mostrados na Figura 6.2. Sequencialmente, acontecem as seguintes transmissões: uma aplicação envia um *returnResponsePdu-SDU* para o Despachante; então o Despachante envia um *prepareResponseMsgIn-SDU* para um modelo de processamento de mensagens e recebe um *prepareResponseMsgOut-SDU* deste mesmo modelo. Por último, o Despachante envia um *Network-MSG* para a rede.

Nenhuma análise é efetuada após a recepção do *returnResponsePdu-SDU*. Esta tupla é então propagada como um *prepareResponseMsgIn-SDU* para o modelo de processamento de mensagens indicado no SDU. Apesar de serem idênticos, este dois SDUs possuem nomes diferentes para enfatizar os dados de qual primitiva[5] eles transmitem. Sua semelhança se deve à especificação informal.

A seguir será definida a ação procedure Disp send-response B, que é responsável por montar a resposta que será enviada para o Despachante de uma entidade externa. Ela corresponde ao

[5]Definida em [HPW99].

estágio B dentro do Despachante (Figura 6.2):

- procedure Disp send-response B :: action[receiving prepareResponseMsgOut-SDU]

(6) procedure Disp send-response B =

```
| | check (the given Result#1 is success)
| or
| | | check not (the given Result#1 is success)
| | and then
| | | send the given Result#1 back to sender
| | and then
| | | escape
and then
| give (the given transportDomain#2, the given transportAddress#3, the given Network-MSG#4)
```

No estágio de análise (do topo da ação até o and then principal), *Result* é comparado com *success*. Sucesso significa que nenhum erro ocorreu no procedimento seguido pelo modelo de processamento de mensagens. Neste caso, a ação passa para o estágio de tradução (a última linha da ação), onde todos os dados recebidos (*prepareOutgoingMsgOut-SDU*) são propagados.

6.4 Procedimento Receive

Este procedimento é executado dentro de um despachante quando uma mensagem é recebida de outra entidade e deve ser enviada a uma aplicação local.

A ação procedure Disp receive é definida como uma seqência, na qual:

1. O sorte dos dados recebidos é checado, e deve ser um *Network-MSG* provindo de outra entidade.
2. Uma comunicação com o agente da semântica de ações executando um modelo de processamento de mensagens dentro da própria entidade é estabelecida, baseado no valor do campo *messageProcessingModel* extraído do *Network-MSG* recebido.
3. Uma aplicação é escolhida para processar o pacote recebido, distinguindo-se dois casos: um quando o pacote recebido conter uma requisição; outro quando conter uma resposta.

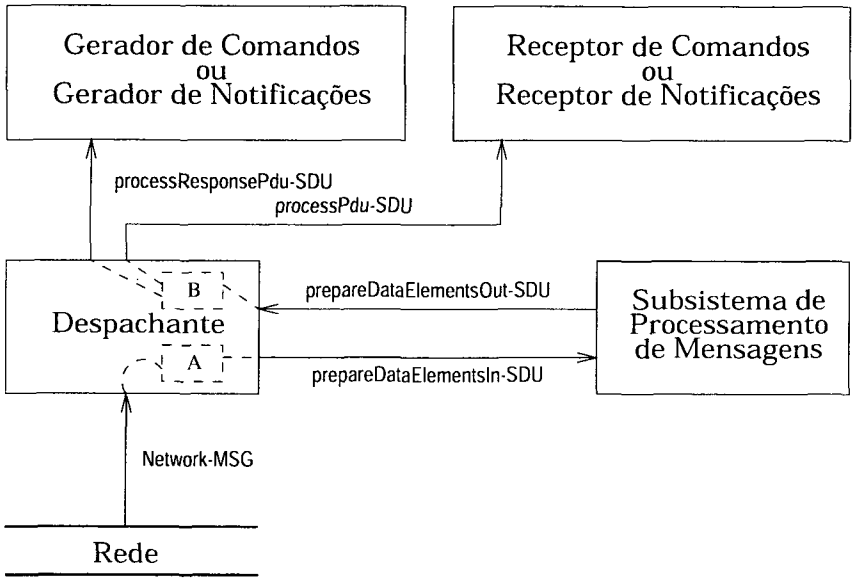

Figura 6.3: Procedimento de recepção de uma requisição ou resposta e troca dos SDUs.

```
• procedure Disp receive :: action[receiving tuple]
(7) procedure Disp receive =
        | give (the given tuple)[Network-MSG]
      then
        | procedure Disp receive A
      then
        | send a message[to the MPModel associated with the given messageProcessingModel#1]
                        [containing (the rest of the given tuple)[prepareDataElementsIn-SDU]]
    and then
        | accept contents of a message[from an MPModel][containing a tuple[prepareDataElementsOut-SDU]]
      then
        | | procedure Disp receive-request B
        then
        | | send a message[to the given Application#1]
                        [containing (the rest of the given tuple)[processPdu-SDU]]
        or
        | | procedure Disp receive-response B
        then
        | | send a message[to the given Application#1]
                        [containing (the rest of the given tuple)[processResponsePdu-SDU]]
```

Os dados trocados entre os módulos da entidade quando uma resposta é recebida são mostrados na Figura 6.3. Sequencialmente, acontecem as seguintes transmissões: o Despachante recebe um *Network-MSG* da rede; então o Despachante envia um *prepareDataElementsIn-SDU* para um modelo de processamento de mensagem e recebe um *prepareDataElementsOut-SDU* deste mesmo modelo. Por último, dependendo do tipo de operação, uma aplicação recebe um *processPdu-SDU* do Despachante(se for uma requisição) ou recebe um *processResponsePdu-SDU* (se for uma resposta).

A seguir será definida a ação procedure Disp receive A, que é responsável por preparar os dados recebidos que serão passados ao modelo de processamento de mensagem da entidade. Ela

corresponde ao estágio A dentro do Despachante (Figura 6.3):

- procedure Disp receive A :: action[receiving Network-MSG]

(8) procedure Disp receive A =

```
| increment("snmpInPkts")
and then
| | | give the messageProcessingModel extracted from the given Network-MSG
  trap
| | | increment("snmpInASNParseErrs")
  | | and then
  | | | escape
then
| | give (the MPModel associated with the given messageProcessingModel)
  or
| | | check not (the MPModel associated with the given messageProcessingModel is an agent)
  | | and then
  | | | increment("snmpInBadVersions")
  | | and then
  | | | escape
and then
| give (the transportDomain extracted from the given Network-MSG,
|       the transportAddress extracted from the given Network-MSG)
and then
| regive
```

No estágio de análise (do topo da ação até o último and then), o objeto *snmpInPkts* do grupo da MIB II [CMRW96a] é incrementada[6]. Então a ação the MPModel associated with _, recebendo com parâmetro um *Network-MSG*, extrai o modelo de processamento de mensagens do *Network-MSG* fornecido. Caso esta ação escape, o objeto *snmpInASNParseErrs* do grupo da MIB II [CMRW96a] é incrementada. Se um agente da semântica de ações não estiver associado ao modelo de processamento de mensagens extraído, então o objeto *snmpInBadVersions* do grupo da MIB II [CMRW96a] é incrementada. Caso contrário, segue-se o estágio de tradução, que consiste na determinação de *transportDomain* e *transportAddress* e na simples propagação do *Network-MSG* recebido.

A seguir serão definidas duas ações: procedure Disp receive-request B e procedure Disp receive-response B. A primeira é responsável por repassar uma requisição recebida para a aplicação correta e a última é responsável no caso de respostas recebidas. A ação procedure Disp receive-request B corresponde a uma parte do estágio B dentro do Despachante (Figura 6.3), como mostrado a seguir:

- procedure Disp receive-request B :: action[receiving prepareDataElementsOut-SDU]

(9) procedure Disp receive-request B =

[6] A ação increment é definida na Seção 6.5.

```
| | | check not (the given Result#1 is sucess)
| | and then escape
| or
| | | check (the given Result#1 is sucess)
| | and then
| | | check (the given sendPduHandle#11 is none)
| | and then
| | | | give the Application associated with (the given contextEngineID#7, the given pduType#10)
| | | trap
| | | | | increment("snmpUnknownPDUHandlers")
| | | | and then
| | | | | | regive and get("snmpUnknownPDUHandlers")
| | | | | then
| | | | | | give (the given messageProcessingModel#2, securityModel#3, securityName#4,
| | | | | |       securityLevel#5, pduVersion#6, contextEngineID#7, contextName#8, PDU#9,
| | | | | |       maxSizeResponseScopedPdu#12, stateReference#14,
| | | | | |       par of(noApplication, the given ObjectValue#15))
| | | | | then
| | | | | | | procedure Disp send-response
| | | | | and then
| | | | | | escape
and then
| give (the given messageProcessingModel#2, securityModel#3, securityName#4, securityLevel#5,
|       pduVersion#6, contextEngineID#7, contextName#8, PDU#9,
|       maxSizeResponseScopedPdu#12, stateReference#14)
```

Na ação procedure Disp receive-request B o campo *Result* recebido deve ser checado para se certificar que o processo efetuado pelo modelo de processamento de mensagens foi bem sucedido. Caso não seja, a ação escapa e todo o processo é abandonado. Então *sendPduHandle* é checado e deve ser *none*. Caso seja, isto indica que uma requisição está sendo recebida e o processo continua. Caso contrário, uma resposta está sendo recebida; então a ação falha e a ação procedure Disp receive-response B será testada.

Quando uma aplicação vai processar operações de gerência, ela deve antes se cadastrar na entidade em que ela atuará. Como se trata de uma requisição recebida, deve existir uma aplicação já cadastrada para processá-la, e esta será obtida pelo produtor the Application associated with, ao se fornecer o tipo da operação e o contexto em que esta será operada.

Caso não exista uma aplicação preparada para processar a requisição recebida, o objeto *snmpUnknownPDUHandlers* do grupo da MIB II é incrementado e uma resposta é preparada para ser enviada de volta à entidade que originou a requisição. A indicação deste erro para este caso é *noApplication*. Para o campo *statusInformation* é então atribuído um par, contendo a indicação de erro acima e o valor do objeto *"snmpUnknownPDUHandlers"* antes incrementado, pelo produtor par of.

Caso tudo proceda bem, uma tupla do tipo *processPdu-SDU* é preparada (última linha).

A ação procedure Disp receive-response B corresponde a uma parte do estágio B dentro do Despachante (Figura 6.3), como mostrado a seguir:

- procedure Disp receive-response B :: action[receiving prepareDataElementsOut-SDU]

```
(10) procedure Disp receive-response B =
        | | | check not (the given Result#1 is sucess)
        | | and then
        | | | escape
        | or
        | | | check (the given Result#1 is sucess)
        | | and then
        | | | check not (the given sendPduHandle#11 is none)
        | | and then
        | | | | give the Application associated with the given sendPduHandle#11
        | | | trap
        | | | | | increment("snmpUnknownPDUHandlers")
        | | | | and then
        | | | | | escape
        and then
        | give (the given messageProcessingModel#2, securityModel#3, securityName#4, securityLevel#5,
        |       pduVersion#6, contextEngineID#7, contextName#8, PDU#9, statusInformation#13,
        |       sendPduHandle#11)
```

Na ação procedure Disp receive-response B o campo *Result* recebido deve ser checado para se certificar que o processo efetuado pelo modelo de processamento de mensagens foi bem sucedido. Caso não seja, a ação escapa e todo o processo é abandonado. Então *sendPduHandle* é checado e deve ser diferente de *none*, indicando a recepção de uma resposta.

A aplicação para a qual se destina a resposta recebida será obtida pelo produtor the Application associated with, ao se fornecer o *sendPduHandle* que veio na mensagem. Este valor é o mesmo que foi antes gerado em uma requisição enviada, para a qual esta é a respectiva resposta.

Caso não exista uma aplicação indicada para processar a resposta recebida, o objeto *snmpUnknownPDUHandlers* do grupo da MIB II é incrementado e o processo é abandonado.

Caso tudo proceda bem, uma tupla do tipo *processResponsePdu-SDU* é preparada (última linha).

6.5 Notação Auxiliar

Esta seção contém definições de dados, ações e produtores auxiliares utilizados na especificação do Despachante SNMP, incluindo abreviaturas e entidades semânticas.

6.5.1 Ações

A ação increment(N) incrementa o valor de um objeto de gerência de nome N. Também pode ser utilizada para incrementar um valor contido numa célula de memória. Em ambos os casos, este valor deve ser numérico e discreto.

- increment(_) :: token → action

```
(11) increment(N:ObjectName) =
        | get(N)
        then
        | set(N, sum(it, 1))
```

(12) increment(K:token) =
store the sum(the integer stored in the cell bound to K, 1) in the cell bound to K

A ação accept foi definida na Seção 5.4.

A ação send _ back to sender é uma abreviatura para o envio de uma tupla para o remetente da requisição original, que é identificado por *"sender"*, e é definida como segue:

- send _ back to sender :: tuple → action

(13) send T:tuple back to sender = send a message[to the cached "sender"][containing T]

A especificação da ação generate sendPduHandle gera rótulos para indexar as mensagens que transitam dentro do engenho e para associá-la a outra mensagem de resposta. Um caso parecido já foi tratado com generate request-id na Seção 5.4. A ação generate sendPduHandle também é dependente da implementação.

- generate sendPduHandle :: action[giving sendPduHandle]

(14) generate sendPduHandle = □

6.5.2 Produtores

O produtor the generated D retorna um índice do sorte D passado como parâmetro. O produtor the generated sendPduHandle retorna o índice *sendPduHandle* do atual procedimento seguido pelo Despachante e sua implementação depende da ação generate sendPduHandle.

- the generated sendPduHandle :: yielder[of sendPduHandle]

(15) the generated sendPduHandle = □

Os seguintes produtores são dependentes da implementação e são relacionados com a extração de dados de uma mensagem SNMP. Três casos são definidos, para extrair respectivamente *messageProcessingModel*, *transportDomain* e *transportAddress*.

- the _ extracted from _ :: tuple, tuple → yielder[of tuple]

(16) the $D \leq$ messageProcessingModel extracted from N:Network-MSG = □

(17) the $D \leq$ transportDomain extracted from N:Network-MSG = □

(18) the $D \leq$ transportAddress extracted from N:Network-MSG = □

Há quatro novos casos de especificação do produtor the A associated with D usados neste capítulo (um caso já foi definido no capítulo anterior).

No primeiro caso, o indicador de um agente da semântica de ações que representa um Despachante é retornado de acordo com o valor do sorte *transportData* fornecido. No segundo caso, o indicador de um agente da semântica de ações que representa um modelo de processamento de mensagens é retornado de acordo com o valor do sorte *messageProcessingModel* fornecido.

No terceiro caso, o indicador de um agente da semântica de ações que representa uma aplicação é retornado de acordo com o valor da tupla fornecida, contendo o tipo da operação e o seu contexto. No quarto caso, o indicador de um agente da semântica de ações que representa uma aplicação é retornado de acordo com o valor do sorte *sendPduHandle* fornecido.

Sempre que não existir uma associação, ou seja, não existir um agente da semântica de ações processando o módulo requerido, o produtor falha.

- the _ associated with _ :: agent, tuple → yielder[of agent]

(19) the $A \leq$ Dispatcher associated with N:transportData = □

(20) the $A \leq$ MPModel associated with M:messageProcessingModel = □

(21) the $A \leq$ Application associated with (C:contextEngineID, O:pduType) = □

(22) the $A \leq$ Application associated with H:sendPduHandle = □

6.5.3 Dados

As primitivas definidas em [HPW99] descrevem a comunicação entre os componentes de um engenho SNMP, e possuem diversos campos de dados, que são listados a seguir. Os itens definidos como □ são desconhecidos neste estágio da especificação.

Dados associados com o envio de uma requisição:

(23) sendPdu-SDU = (transportData, messageData, securityData, accessData, expectResponse)

(24) prepareOutgoingMsgIn-SDU = (transportData, messageData, securityData, accessData, expectResponse, sendPduHandle)

(25) prepareOutgoingMsgOut-SDU = (statusInformation, transportData, Network-MSG)

Dados associados com o envio de uma resposta:

(26) returnResponsePdu-SDU = (messageData, securityData, accessData, maxSizeResponseScopedPdu, stateReference, statusInformation)

(27) prepareResponseMsgIn-SDU = (messageData, securityData, accessData, maxSizeResponseScopedPdu, stateReference, statusInformation)

(28) prepareResponseMsgOut-SDU = (Result, transportData, Network-MSG)

Dados associados com a recepção de operações:

(29) prepareDataElementsIn-SDU = (transportData, Network-MSG)

(30) prepareDataElementsOut-SDU = (Result, messageData, securityData, accessData, pduType, sendPduHandle, maxSizeResponseScopedPdu, statusInformation, stateReference)

(31) processPdu-SDU = (messageData, securityData, accessData, maxSizeResponseScopedPdu, stateReference)

(32) processResponsePdu-SDU = (messageData, securityData, accessData, statusInformation, sendPduHandle)

Dados transmitidos pela rede:

(33) Network-MSG = v3MPMessage | □

O sorte SDU é composto por todas as unidades de dados trocadas entre os módulos de uma entidade SNMP e pelas mensagens que fluem pela rede.

(34) SDU $\geq$ sendPdu-SDU | prepareOutgoingMsgIn-SDU | prepareOutgoingMsgOut-SDU | returnResponsePdu-SDU | prepareResponseMsgIn-SDU | prepareResponseMsgOut-SDU | prepareDataElementsIn-SDU | prepareDataElementsOut-SDU | processPdu-SDU | processResponsePdu-SDU | Network-MSG

Os seguintes sortes representam os dados usados por cada subsistema dentro de um engenho SNMP, especificando os parâmetros usados pela rede, mensagem, segurança e PDU dentro dos SDUs.

(35) transportData = (transportDomain, transportAddress)

(36) messageData = (messageProcessingModel)

(37) securityData = (securityModel, securityName, securityLevel)

(38) accessData = (pduVersion, scopedPdu)

Os campos que compõem os SDUs e não foram mencionados aqui (i.e., *messageProcessingModel*) fazem parte da especificação informal e são definidos juntamente com as primitivas em [HPW99], seguindo o estilo apresentado naquele documento. Algumas novas definições são introduzidas abaixo para melhorar a clareza da especificação do SNMP.

Para *errorIndication*, neste capítulo somente foram definidas situações de erro quanto ao modelo de mensagem não conhecido pelo despachante (*noMPModel*) e quanto à ausência de uma aplicação para processar uma requisição recebida (*noApplication*). Os valores aqui usados para *errorIndication* são simbólicos e apenas diferenciam sua ocorrência.

(39) handle $\geq$ sendPduHandle

(40) sendPduHandle = none — $\square$ (*individual*)

(41) handle $\geq$ stateReference

(42) pduType = opTag

(43) errorIndication = noMPModel | noApplication | $\square$ (*individual*)

(44) par of (E:errorIndication, V:ObjectValue) $\rightarrow$ errorIndication

(45) statusInformation = Result = success | errorIndication

(46) agent $\geq$ Application | MPModel | Dispatcher

6.6 Considerações Finais

Este capítulo tratou da definição formal dos procedimentos seguidos pelo Despachante SNMP. O próximo capítulo trata das aplicações padrão do SNMP, que geram, enviam, recebem e processam as requisições propagadas pelo Despachante. Todo objeto de MIB utilizado neste capítulo, referente às informações salvas temporariamente, será definido no Capítulo 8. Lá também será mostrado como o Despachante é inserido dentro da especificação de uma entidade completa.

CAPÍTULO 7

Aplicações SNMP

Este capítulo continua a especificação da entidade SNMP iniciada no capítulo anterior, tratando da especificação formal das aplicações padrão definidas pelo protocolo SNMPv3: Gerador de Comandos, Receptor de Comandos, Gerador de Notificações, Receptor de Notificações e *Proxy Forwarder*. A Seção 7.1 trata da especificação formal da aplicação Gerador de Comandos, a Seção 7.2 trata da especificação formal da aplicação Receptor de Comandos, a Seção 7.3 trata da especificação formal da aplicação Gerador de Notificações e a Seção 7.4 trata da especificação formal da aplicação Receptor de Notificações. A Seção 7.5 aborda as ações auxiliares utilizadas nesta descrição. Os SDUs que são trocados pelas aplicações padrão são os utilizados pelo Despachante, como mostrado na Seção 6.5.

Este capítulo é uma extensão de [FMD00a], considerando a existência do Despachante descrito no capítulo anterior dentro da entidade, além dos outros módulos do engenho. Toda a especificação a seguir é efetuada a partir da especificação informal das aplicações padrão em [LMS99].

A especificação em semântica de ações das aplicações padrão é escrita de forma *top-down*.

7.1 Gerador de Comandos

Uma aplicação do tipo Gerador de Comandos é responsável por gerar e enviar requisições contendo operações de leitura e escrita, e por receber e processar sua devida resposta.

A seguinte ação corresponde ao procedimento seguido por uma aplicação Gerador de Comandos:

- procedure CommandGenerator :: action

(1) procedure CommandGenerator =

```
| | procedure Aplic send-request
| then
| | send a message[to Dispatcher of entity][containing (the given tuple)[sendPdu-SDU]]
| then
| | accept contents of a message[from Dispatcher of entity][containing a datum]
| then
| | | ifnot (it is a sendPduHandle) generate error it
| | | and then
| | | | cache it as "sendPduHandle"
and then
| | accept contents of a message[from Dispatcher of entity]
| |                                [containing a tuple[processResponsePdu-SDU]]
| then
| | procedure Aplic receive-response
```

O envio da requisição e a espera da respectiva resposta são ações executadas sequencialmente, e por isto separadas por um combinador and then (combinador principal da ação).

Caso a operação a ser gerada seja *GetBulk*, os dados recebidos como informação transitória devem corresponder a *"operation-type"*, uma lista de nomes de variáveis ou uma *VarBindList*, *"non-repeaters"* e *"max-repetitions"*. Caso a operação seja qualquer outra, os dados recebidos como informação transitória devem corresponder somente a *"operation-type"* e uma lista de nomes de variáveis ou uma *VarBindList*. Estes dados são passados para a ação procedure Aplic send-request, que vai gerar um *sendPdu-SDU* a ser enviado para o Despachante da entidade. O Despachante é obtido através do produtor Dispatcher of entity[1], que retorna o agente da semântica de ações responsável por esta função na atual entidade.

Uma confirmação do envio do *sendPdu-SDU* é então esperada, contendo ou o índice que irá controlar a recepção da respectiva resposta (quando a mensagem é enviada com sucesso), ou a informação de um erro que impossibilitou o envio da mensagem pelo Despachante. A ação ifnot [2], testa o valor recebido, permitindo que este valor seja armazenado pela ação cache para correlacionar esta requisição com sua respectiva resposta quando a mensagem tiver sido enviada com sucesso (*sendPduHandle*) ou escapando com este valor, quando for uma indicação de erro (*errorIndication*). Todos os procedimentos seguidos quando a ação procedure CommandGenerator termina excepcionalmente serão tratados na aplicação usuária que chamou-a, pois as ações a serem tomadas em cada caso são dependentes da implementação.

A respectiva resposta, que é recebida através de um *processResponsePdu-SDU*, é então aguardada. Quando recebida, a ação procedure Aplic receive-response testa diversos valores enviados com os recebidos para validar se a resposta não foi aceita por engano ou foi violada. Caso aprovado, os campos do PDU recebido são retornados para a aplicação usuária que requisitou a operação, que então deverá executar uma ação dependente da implementação para

[1] Sua definição pode ser vista no Capítulo 8, quando uma entidade é especificada com seus módulos.

[2] Definida na Seção 5.4.

processar os dados recebidos ou para processar o erro ocorrido, que poderá tanto decorrer da terminação excepcional da ação procedure CommandGenerator como de um erro ocorrido durante o processamento da requisição e indicado pelo campo *"statusInformation"*. Esta ação será então a retransmissão da requisição ou a notificação (de outra aplicação gerente ou de uma pessoa operando a aplicação) sobre a falha ocorrida.

A seguir é especificada a ação procedure Aplic send-request, que tem a função de formar o *sendPdu-SDU* que será enviado para o Despachante da entidade:

```
• procedure Aplic send-request :: action[giving sendPdu-SDU]
(2) procedure Aplic send-request =
        give (the cached "transpDom", the cached "transpAdd", the cached "MPModel",
              the cached "secModel", the cached "secName", the cached "secLevel",
              the cached "pduVersion", the cached "contEngID", the cached "contName")
      and
        check not (the given opTag#1 is GetBulk)
        and then
        operationRequest the given opTag#1 with (0, 0, the given list#2)
      or
        check (the given opTag#1 is GetBulk)
        and then
        operationRequest GetBulk with (the given non-repeaters#3, the given max-repetitions#4,
                                       the given list#2)
      and
      give not (the given opTag#1 is Trap)
```

A ação procedure Aplic send-request utiliza os valores previamente armazenados de *"transpDom"*, *"transpAdd"*, *"MPModel"*, *"secModel"*, *"secName"*, *"secLevel"*, *"pduVersion"*, *"contextEngID"* e *"contextName"*.

O PDU a ser enviado é formado pela ação operationRequest [3]. No caso da operação *GetBulk*, os valores referentes a *"non-repeaters"* e *"max-repetitions"* devem ser fornecidos como informação transitória. Em todos os casos, *"operation-type"* e *"variable-bindings"* provêm de valores recebidos como informação transitória.

Na última linha, o valor para *"expectResponse"* é *false* quando a operação for *Trap* e *true* caso contrário.

A seguir é especificada a ação procedure Aplic receive-response, que é executada quando a resposta da operação é recebida:

```
• procedure Aplic receive-response :: action[receiving processResponsePdu-SDU]
(3) procedure Aplic receive-response =
```

[3]Definida na Seção 5.2.1.

```
| | ifnot (the given messageProcessingModel#1 is the cached "MPModel")
| |       generate error discardByMPModel
| and
| | ifnot (the given securityModel#2 is the cached "secModel")
| |       generate error discardBySecModel
| and
| | ifnot (the given securityName#3 is the cached "secName")
| |       generate error discardBySecName
| and
| | ifnot (the given pduVersion#5 is the cached "pduVersion")
| |       generate error discardByPduVersion
| and
| | ifnot (the given contextEngineID#6 is the cached "contEngID")
| |       generate error discardByContextEngID
| and
| | ifnot (the given contextName#7 is the cached "contName")
| |       generate error discardByContextName
| and
| | ifnot (the request-id of the given Pdu#8 is the generated "request-id")
| |       generate error discardByRequestID
and then
| give (error-status of the given Pdu#8, error-index of the given Pdu#8,
|       variable-bindings of the given Pdu#8)
```

A ação procedure Aplic receive-response testa se os valores enviados e os valores recebidos dos campos *messageProcessingModel*,*securityModel*, *securityName*, *pduVersion*, *contextEngineID*, *contextName* e *"request-id"* são os mesmos e caso todos os testes coincidam, o PDU é quebrado e os valores de seus campos *"error-status"*, *"error-index"* e *"variable-bindings"* são retornados como informação transitória. O produtor the generated request-id [4] retorna o *"request-id"* previamente gerado ao se enviar a requisição.

Os valores *discardByMPModel*, *discardBySecModel*, *discardBySecName*, *discardByContextEngID*, *discardByContextName*, *discardByPduVersion* e *discardByRequestID* são indicações de erro e são dependentes da implementação.

7.2 Receptor de Comandos

Uma aplicação do tipo Receptor de Comandos é responsável por receber e processar requisições contendo operações de leitura e escrita, e por gerar e enviar sua devida resposta.

A seguinte ação corresponde ao procedimento seguido por uma aplicação Receptor de Comandos:

- procedure CommandResponder :: action

(4) procedure CommandResponder =

[4] Definido na Seção 7.5.

```
| accept contents of a message[from Dispatcher of entity][containing a tuple[processPdu-SDU]]
then
| procedure Aplic receive-request
then
| | give the cached "variable-bindings" then map using abstraction of isAccessAllowed
| and then
| | operationResponse the given opTag#5
then
| procedure Aplic send-response
then
| send a message[to Dispatcher of entity][containing (the given tuple)[returnResponsePdu-SDU]]
```

Uma mensagem contendo um *processPdu-SDU* é recebida. Então, a ação procedure Aplic receive-request recebe este SDU como informação transitória, processa-o salvando algumas informações e devolvendo os campos do PDU recebido como informação transitória.

Antes que o processamento da operação inicie, a ação isAccessAllowed[5] será executada para cada variável recebida, a fim de avaliar se todas as variáveis estão dentro de um contexto que permita que elas sejam operadas. Este contexto será avaliado pelo Subsistema de Controle de Acesso da entidade, através da ação antes citada. A seguir, a operação de gerência é executada pela ação operationResponse e um PDU é devolvido. Este PDU é passado para a ação procedure Aplic send-response, que vai montar um *returnResponsePdu-SDU* a ser enviado como resposta.

A ação isAccessAllowed verifica se a entidade que enviou a atual *VarBind* tem acesso para realizar a operação de gerência de redes que deseja efetuar. Para isto uma visão da MIB contendo a entidade e o objeto de gerência mencionado é testada, para que a operação possa ser validada ou não. Este processo consiste em enviar-se uma mensagem para o Subsistema de Controle de Acesso a fim de que se realize esta consulta. Então, sua resposta indica a forma de acesso disponível para esta variável, terminando normalmente ou excepcionalmente dependendo deste resultado. O produtor class of *0* retorna a classe de uma operação de gerência.

- isAccessAllowed :: action

(5) isAccessAllowed =

[5]Definida na Seção 7.5.

```
| give (the cached "secModel", the cached "secName", the cached "secLevel",
|       class of the cached "pduType", the cached "contextName", the given ObjectName#1)
then
| send a message[to ACModel of entity][containing then]
then
| accept contents of message[from ACModel of entity][containing an StatusInformation]
then
| | | check all (it is noSuchView, it is noAccessEntry, it is noGroupNome)
| | and then
| | | | setError(authorizationError, 0) and give the cached "variable-bidings"
| | | then escape with them
| | else
| | | check (it is noSuchContext)
| | and then
| | | | setError(0,0) and give empty-list
| | | then escape with them
| | else
| | | setError(genError,0) and give the cached "variable-bidings"
| | then escape with them
```

A seguir é especificada a ação procedure Aplic receive-request, que é executada para processar a operação que foi recebida:

- procedure Aplic receive-request :: action[receiving processPdu-SDU][giving (request-id,error-status,Index,list,opTag)]

(6)
```
procedure Aplic receive-request =
| ifnot (the performing-agent is associated with the operation of the given PDU#8)
|       generate error discardByOperation
and then
| | give (request-id of the given PDU#8, error-status of the given PDU#8,
| |       error-index of the given PDU#8, variable-bindings of the given PDU#8,
| |       operation of the given PDU#8)
| then
| | | regive and cache the splitted PDU
and then
| | cache the given messageProcessingModel as "MPModel"
| and
| | cache the given securityModel as "secModel"
| and
| | cache the given securityName as "secName"
| and
| | cache the given securityLevel as "secLevel"
| and
| | cache the given pduVersion as "pduVersion"
| and
| | cache the given contextEngineID as "contextEngID"
| and
| | cache the given contextName as "contextName"
```

A ação procedure Aplic receive-request é responsável por processar um *processPdu-SDU* recebido. Primeiro é verificado pela ação ifnot[6] se a aplicação atual está associada com o tipo de operação a ser tratada, o que significa validar se a operação pode ser realizada por esta aplicação. O produtor is associated[7] retorna *true* se existir um registro da atual aplicação com a operação a ser processada.

Caso verdadeiro, o PDU recebido é desempacotado, e então devolvido como informação

[6]Definida na Seção 7.5.

[7]Definido na Seção 7.5.

transitória e salvo para posterior processamento pela ação cache the splitted PDU. As outras informações do SDU, que serão utilizadas durante a geração da resposta, são então salvas pela ação cache, incluindo *"MPModel"*, *"secModel"*, *"secName"*, *"secLevel"*, *"pduVersion"*, *"contextEngID"* e *"contextName"*.

A seguir é especificada a ação procedure Aplic send-response, que é executada para formar a resposta da operação anteriormente recebida e processada:

- procedure Aplic send-response :: action[giving returnResponsePdu-SDU]

```
(7)  procedure Aplic send-response =
         | give (the cached "MPModel",
         |       the cached "secModel", the cached "secName", the cached "secLevel",
         |       the cached "pduVersion", the cached "contEngID", the cached "contName")
       and then
         | give Pdu of (the cached "request-id", the given errorStatus#1,
         |               the given Index#2, the given VarBindList#3) with tag Response
       and then
         | give (size of scopedPdu, the cached "stateReference", the given errorStatus#1)
```

A ação procedure Aplic send-response monta um *returnResponsePdu-SDU* a ser enviado para o Despachante. Para tanto, os valores de *"MPModel"*, *"secModel"*, *"secName"*, *"secLevel"*, *"pduVersion"*, *"contextEngID"* e *"contextName"* são recuperados; um PDU é então empacotado, recebendo uma tupla contendo um *errorStatus*, um *Index* e um VarBindList da ação operationResponse, executada anteriormente à ação procedure Aplic send-response; e o valor retornado como *statusInformation* será *success* ou um *errorIndication*, de acordo com o processamento efetuado com o PDU. O produtor size of scopedPdu atribui o valor de *"maxSizeResponseScopedPDU"* de forma dependente da implementação.

7.3 Gerador de Notificações

Uma aplicação do tipo Gerador de Notificações é responsável por gerar e enviar requisições contendo operações de notificação, e por receber e processar sua devida resposta.

A seguinte ação corresponde ao procedimento seguido por uma aplicação Gerador de Notificações:

- procedure NotificationOriginator :: action

```
(8)  procedure NotificationOriginator =
```

```
| | give the given list#2
| then
| | map using abstraction of isAccessAllowed
and then
| | procedure Aplic send-request
| then
| | send a message[to Dispatcher of entity][containing (the given tuple)[sendPdu-SDU]]
and then
| | | check (the given opTag#1 is Inform)
| | and then
| | | | accept contents of a message[from Dispatcher of entity]
| | | |                              [containing a tuple[processResponsePdu-SDU]]
| | | then
| | | | give (the error-status of the given Pdu#8, the error-index of the given Pdu#8,
| | | |       the given variable-bindings of the given Pdu#8)
| or
| | check (the given opTag#1 is Trap)
```

O tipo da operação e uma lista são recebidos como informação transitória, sendo que esta lista deve ser uma *VarBindList* ou uma lista de nomes de variáveis. A ação isAccessAllowed [8] é usada para validar o acesso dos dados a serem enviados.

Estes dados são passados como informação transitória para a ação procedure Aplic send-request, que vai gerar um *sendPdu-SDU* a ser enviado para o Despachante da entidade. Nenhuma confirmação é esperada para se saber se a mensagem foi enviada com sucesso ou não, apesar de enviada pelo Despachante. Então, no caso em que a operação enviada foi *Inform*, um *processResponsePdu-SDU* é esperado, e quando recebido, apenas os valores de *"error-status"*, *"error-index"* e *"variable-bindings"* contidos no PDU são retornados para a aplicação usuária que iniciou a requisição. Então, esta aplicação usuária irá executar uma ação dependente da implementação para processar os dados recebidos ou para processar o erro ocorrido, da mesma forma que na aplicação Gerador de Comandos.

Somente uma parte da especificação informal da aplicação Gerador de Notificações foi formalizada na ação procedure NotificationOriginator Foram feitas restrições com relação à suspensão da recepção após um intervalo de tempo e com relação à retransmissão de mensagens neste caso, pois não há uma ação que permita aguardar uma mensagem só por um determinado tempo em semântica de ações[9].

A ação procedure Aplic send-request foi definida na Seção 7.1.

[8] Definida na Seção 7.2.

[9] A inexistência de mecanismos de controle de tempo é uma característica da faceta comunicativa, sendo um tema de pesquisa ainda aberto.

7.4 Receptor de Notificações

Uma aplicação do tipo Receptor de Notificações é responsável por receber e processar requisições contendo operações de notificação, e por gerar e enviar sua devida resposta.

A seguinte ação corresponde ao procedimento seguido por uma aplicação Receptor de Notificações:

- procedure NotificationReceiver :: action

(9) procedure NotificationReceiver =
| accept contents of a message[from Dispatcher of entity][containing a tuple[processPdu-SDU]]
then
| procedure Aplic receive-request
then
| | | check (the given opTag#5 is Inform)
| | and then
| | | | operationResponse Inform
| | | then
| | | | procedure Aplic send-response
| | | then
| | | | send a message[to Dispatcher of entity][containing (the given tuple)[returnResponsePdu-SDU]]
| | or
| | | | check (the given opTag#5 is Trap)
| | and then
| | | | operationResponse Trap

Uma mensagem contendo um *processPdu-SDU* é recebida. Então, a ação procedure Aplic receive-request recebe este SDU como informação transitória, processa-o salvando algumas informações e devolve os campos do PDU recebido como informação transitória. Caso a operação a ser executada seja *Trap*, isto é feito e o processo termina. Caso a operação seja *Inform*, ela é executada e, em seguida, os dados por ela devolvidos são utilizados pela ação procedure Aplic send-response, que vai montar um *returnResponsePdu-SDU* a ser enviado como resposta pelo Despachante da entidade.

As ações procedure Aplic receive-request e procedure Aplic send-response foram definidas na Seção 7.2.

7.5 Notação Auxiliar

Esta seção contém definições de dados, ações e produtores auxiliares utilizados na especificação das aplicações padrão do SNMP, incluindo abreviaturas e entidades semânticas.

7.5.1 Ações

Uma nova forma da ação ifnot X generate error E é dada nesta seção[10], usada quando o erro E a ser retornado for do sorte *errorIndication*. Seus dois parâmetros são um produtor de

[10] Veja também a Seção 5.4.

valor-verdade e um dado. Caso o primeiro argumento resulte em falso, a ação termina excepcionalmente passando o segundo argumento. Caso contrário, a ação termina normalmente. Ela é útil para simplificar a escrita de múltiplos testes consecutivos.

- ifnot _ generate error _ :: yielder[of truth-value], errorIndication → action

(10) ifnot *Y*:yielder generate error *E*:errorIndication =
| check *Y*
or
| | check not *Y*
| and then
| | escape with *E*

7.5.2 Produtores

O produtor Dispatcher of entity devolve a referência ao único módulo Despachante da entidade em questão. Sua implementação depende da alocação dos agentes da semântica de ações cuja solução será apresentada no Capítulo 8.

- Dispatcher of entity :: yielder[of agent]

(11) Dispatcher of entity = □

O produtor ACModel of entity devolve a referência a um modelo do Subsistema de Controle de Acesso. Sua implementação depende da alocação dos agentes da semântica de ações e é dependente da implementação neste estágio da especificação.

- ACModel of entity :: yielder[of agent]

(12) ACModel of entity = □

O produtor the generated request-id retorna o índice *"request-id"* da atual requisição produzida e sua implementação depende da ação generate request-id.

- the generated request-id :: yielder[of requestId]

(13) the generated request-id = □

O produtor the *A* is associated with *O* retorna verdadeiro se a operação *O* estiver associada com a aplicação *A* através da ação associate *A* with *D*[11], usada para cadastrar PDUs por uma aplicação.

- the _ is associated with _ :: agent, opTag → yielder[of truth-value]

(14) the *A*:agent is associated with *O*:opTag = □

O produtor class of *0* retorna a classe de uma operação de gerência, que pode ser leitura, escrita ou notificação.

- class of _ :: opTag → yielder

(15) class of *0*:opTag = □

O produtor size of scopedPdu retorna o tamanho máximo de um *scopedPdu* que a aplicação pode aceitar. Este valor é dependente da implementação.

- size of scopedPdu :: yielder[of integer]

(16) size of scopedPdu = □

[11] Definida na Seção 8.4.

7.5.3 Dados

Os SDUs utilizados neste capítulo já foram mencionados na Seção 6.5.

Somente o sorte *errorIndication* possui novos elementos que são usados para indicar a recusa de uma mensagem recebida por alguma inconsistência nos seus dados. Outros dados auxiliares são mostrados a seguir.

(17) errorIndication = □ | discardByMPModel | discardBySecModel | discardBySecName | discardByContextEngID | discardByContextName | discardByPduVersion | discardByRequestID | discardByOperation (*individual*)

(18) errorStatus = □ | authorizationError (*individual*)

(19) success = noError

7.6 Considerações Finais

Este capítulo tratou da definição formal dos procedimentos seguidos pelas aplicações Gerador de Comandos, Receptor de Comandos, Gerador de Notificações e Receptor de Notificações, que são as aplicações padrão do SNMPv3. Todo objeto de MIB utilizado neste capítulo, referente às informações salvas temporariamente, será definido no Capítulo 8. Lá também será mostrado como as aplicações são inseridas dentro da especificação de uma entidade completa.

CAPÍTULO 8

Entidade SNMP

Este capítulo trata do fechamento de uma entidade como um todo, mostrando seus módulos componentes e sua interação a fim de que a entidade exerça sua requerida função. Como vimos no Capítulo 3, uma entidade SNMP é formada por um engenho e um conjunto de aplicações e dependendo da sua composição pode atuar como gerente ou agente.

A Seção 8.1 mostra como as ações que correspondem aos módulos do engenho SNMP (incluindo as ações definidas no Capítulo 6 sobre o módulo Despachante) serão executadas por agentes da semântica de ações. A Seção 8.2 mostra como as ações que correspondem às aplicações padrão do SNMP (definidas no Capítulo 7) serão executadas por agentes da semântica de ações. A Seção 8.3 propõe então dois tipos de entidades SNMP, agente e gerente, e define sua inicialização usando semântica de ações, mostrando quais agentes da semântica de ações serão subordinados, usando as ações definidas nas seções anteriores. As bases de dados locais (LCDs) e outras ações auxiliares são apresentadas na Seção 8.4.

Toda a especificação a seguir é efetuada a partir da especificação informal de uma entidade como mostrado em [HPW99].

8.1 Engenho

Esta seção define os agentes da semântica de ações executando as ações referentes aos módulos do engenho SNMP. Como visto anteriormente, o engenho SNMP é formado por um Despachante, um subsistema de processamento de mensagens, um subsistema de segurança e um subsistema de controle de acesso, sendo que a sua especificação em semântica de ações é organizada como um conjunto de agentes da semântica de ações, cada um responsável por um destes módulos ou um de seus modelos. A comunicação entre eles se dá através da troca de mensagens.

Um agente da semântica de ações executando a ação referente ao Despachante é descrito a seguir. Os outros módulos do engenho não serão especificados neste trabalho, sendo propostos para um trabalho futuro.

```
• Dispatcher-daemon :: action
(1) Dispatcher-daemon =
        | initialize LCD-Dispatcher
      hence unfolding
      | | | procedure Dispatcher
      | | trap complete
      | and then unfold
```

Um Despachante é um agente da semântica de ações executando a ação procedure Dispatcher, que espera por mensagens endereçadas ao Despachante. O combinador unfolding faz com que esta ação seja executada infinitamente, e o agente da semântica de ações passa a atuar como se fosse um servidor com a função de executar esta ação.

A ação trap complete termina normalmente, impedindo a propagação do estado de terminação excepcional, gerado por uma ação interna á ação procedure Dispatcher, o que fará o Despachante aguardar por uma nova mensagem a ser processada. Este tipo de comportamento é necessário quando a ação deve terminar normalmente para que o processamento que vem a seguir (fora do contexto da ação) possa continuar.

Cada módulo de uma entidade pode possuir um conjunto de variáveis locais, para salvar informações temporariamente, definidas na forma de uma MIB. Estas variáveis podem ser utilizadas para salvar informações recebidas numa requisição ou para salvar dados que são utilizados ao se gerar novas mensagens, indicando contextos ou modelos, por exemplo. A única variável local que um Despachante necessita é a variável *"sender"*. A ação initialize LCD-Dispatcher produz a MIB contendo a variável descrita acima.

```
• initialize LCD-Dispatcher :: action
(2) initialize LCD-Dispatcher =
        allocate a cell then bind "sender" to it
```

Os subsistemas de processamento de mensagens, de segurança e de controle de acesso são formados por um ou mais modelos, cada um sendo executado por um agente da semântica de ações. Isto é feito subordinando-se um agente da semântica de ações para executar cada modelo, e por fim passando a ação que este agente da semântica de ações deverá executar (este processo é feito na Seção 8.3).

O modelo já existente para o Subsistema de Processamento de Mensagens é o v3MP, o modelo já existente para o Subsistema de Segurança é o *User-based Security Model* (USM) e o modelo já existente para o Subsistema de Controle de Acesso é o *View-based Access Control Model* (VACM). Suas especificações formais ficam propostas como trabalho futuro.

8.2 Aplicações

Esta seção define os agentes da semântica de ações executando as ações referentes às aplicações padrão do SNMP. O conjunto de aplicações de uma entidade envolve aplicações do tipo Gerador de Comandos, Receptor de Comandos, Gerador de Notificações, Receptor de Notificações e *Proxy Forwarder*.

As aplicações Receptor de Comandos e Receptor de Notificações serão aqui produzidas como sendo um agente da semântica de ações executando uma ação que corresponde ao procedimento que a aplicação deve seguir.

Já as aplicações Gerador de Comandos e Gerador de Notificações não serão executadas por um agente da semântica de ações, e sim serão chamadas por aplicações usuárias. *Aplicações usuárias* são aplicações não padronizadas, escritas para controlar e monitorar informações de gerência. Isto ocorrerá porque na especificação informal não há nenhuma referência a respeito da comunicação das aplicações Gerador de Comandos e Gerador de Notificações com aplicações usuárias na forma de primitivas. Já as aplicações Receptor de Comandos e Receptor de Notificações possuem esta referência quando, dentro do processamento das operações *Inform* e *Trap*, uma aplicação usuária deve ser escolhida para realizar um processo dependente da implementação em cada caso.

8.2.1 Gerador de Comandos

Este tipo de aplicação não funcionará como um servidor, mas será implementado dentro de uma aplicação usuária que necessite enviar consultas ou alterações para MIBs em outras entidades. Ou seja, não é necessário que um agente da semântica de ações fique responsável por controlar todo o processamento efetuado por esta aplicação. Entretanto, qualquer aplicação usuária que deseje enviar uma operação de leitura ou de escrita irá executar a ação procedure CommandGenerator, que envia uma única requisição e recebe sua respectiva resposta.

Abaixo, vê-se um exemplo de aplicação usuária atuando como emissora de operações SNMP de leitura e de escrita.

- user-application _ :: integer → action

(3) user-application X:integer =
    | initialize LCD-Generate
    hence
    | ...
    | procedure CommandGenerator
    | ...

Inicialmente, a LCD da aplicação Gerador de Comandos é construída pela ação initialize LCD-Generate. Então, em algum ponto da especificação, a ação procedure CommandGenerator é

chamada, recebendo a informação transitória correta, que é constituída por dados referentes ao PDU a ser enviado [1].

As variáveis locais que um Gerador de Comandos necessita são: *"transpDom"*, *"transpAdd"*, *"MPModel"*, *"secModel"*, *"secName"*, *"secLevel"*, *"pduVersion"*, *"contextEngID"*, *"contextName"* e *"request-id"*. A ação initialize LCD-Generate produz a MIB contendo as variáveis descritas acima.

- initialize LCD-Generate :: action

(4) initialize LCD-Generate =
```
    | rebind
    and allocate a cell then bind "transpDom" to it
    and allocate a cell then bind "transpAdd" to it
    and allocate a cell then bind "MPModel" to it
    and allocate a cell then bind "secModel" to it
    and allocate a cell then bind "secName" to it
    and allocate a cell then bind "secLevel" to it
    and allocate a cell then bind "pduVersion" to it
    and allocate a cell then bind "contextEngID" to it
    and allocate a cell then bind "contextName" to it
```

8.2.2 Receptor de Comandos

Esta aplicação será representada por um agente da semântica de ações, executando a seguinte ação:

- CommandResponder-daemon :: action

(5) CommandResponder-daemon =
```
    | | initialize MIBs
    | and
    | | initialize LCD-Process
    | and
    | | register CR
    hence unfolding
    | | | procedure CommandResponder
    | | trap complete
    | and then unfold
```

O processo para inserir novos objetos de gerência em uma entidade é muito simples, bastando especificar como o objeto será atribuído e consultado (ações get e set para um objeto de gerência) e inserir uma célula de memória na ação initialize MIBs, caso necessário. As MIBs de uma entidade podem ser inicializadas logo que uma aplicação Receptor de Comandos começar a executar.

Então, a LCD da aplicação Receptor de Comandos é construída pela ação initialize LCD-Process. As variáveis locais que um Receptor de Comandos necessita são: *"MPModel"*, *"secModel"*, *"secName"*, *"secLevel"*, *"pduVersion"*, *"contextEngID"*, *"contextName"*, *"request-id"*, *"non-repeaters"*, *"max-repetitions"* e *"variable-bindings"*.

- initialize LCD-Process :: action

[1] Veja também a Seção 7.1.

```
(6)  initialize LCD-Process =
         | rebind
         and allocate a cell then bind "MPModel" to it
         and allocate a cell then bind "secModel" to it
         and allocate a cell then bind "secName" to it
         and allocate a cell then bind "secLevel" to it
         and allocate a cell then bind "pduVersion" to it
         and allocate a cell then bind "contextEngID" to it
         and allocate a cell then bind "contextName" to it
         and allocate a cell then bind "request-id" to it
         and allocate a cell then bind "non-repeaters" to it
         and allocate a cell then bind "max-repetitions" to it
         and allocate a cell then bind "variable-bindings" to it
```

Antes que uma aplicação Receptor de Comandos possa processar mensagens, ela deve associar-se com um engenho SNMP, para que os tipos de operações que ela irá processar estejam indicados. A primitiva usada para este propósito é *registerContextEngineID*, sendo formalizada na ação associate A with D, que associa o agente que está executando a ação com o dado D, que contém o identificador do engenho ao qual o agente da semântica de ações está se cadastrando e o respectivo tipo de operação.

É comum que apenas uma aplicação Receptor de Comandos esteja cadastrada por engenho. Uma vez que isto aconteça, ela pode esperar por mensagens do tipo especificado.

```
• register CR :: action
(7)  register CR =
         | get("snmpEngineID")
         then
         | | associate performing-agent with (it, Get)
         | and
         | | associate performing-agent with (it, GetNext)
         | and
         | | associate performing-agent with (it, GetBulk)
         | and
         | | associate performing-agent with (it, Set)
```

Então, a ação procedure CommandResponder é executada eternamente dentro de um *loop*, processando todas as operações de leitura e escrita recebidas pela entidade. A ação trap complete termina normalmente, impedindo que qualquer escape gerado internamente à ação procedure CommandResponder flua para fora do contexto atual, e permitindo que uma nova mensagem seja tratada.

8.2.3 Gerador de Notificações

Da mesma forma como na aplicação Gerador de Comandos, este tipo de aplicação não funcionará como um servidor, mas será implementado dentro de uma aplicação usuária que necessite enviar notificações para outras entidades.

Abaixo, vê-se um exemplo de aplicação usuária atuando como emissora de notificações SNMP.

```
(8)  user-application X:integer =
         | initialize LCD-Generate
       hence
         | ...
         | procedure NotificationOriginator
         | ...
```

Antes de sua execução, a aplicação usuária que for utilizá-la deverá determinar as entidades de destino para as quais uma notificação será enviada, determinar o contexto atual com as informações a serem transmitidas e determinar a lista de variáveis a ser enviada para cada destino. Só então, a ação procedure NotificationOriginator será executada para cada destino, com os respectivos nomes e valores das variáveis passados para a ação como informação transitória.

8.2.4 Receptor de Notificações

Da mesma forma como na aplicação Receptor de Comandos, esta aplicação será representada por um agente da semântica de ações, executando a seguinte ação:

- NotificationReceiver-daemon :: action

```
(9)  NotificationReceiver-daemon =
         | | initialize LCD-Process
         | and
         | | register NR
       hence unfolding
         | | | procedure NotificationReceiver
         | | trap complete
         | and then unfold
```

Inicialmente, a LCD da aplicação Receptor de Notificações é construída pela ação initialize LCD-Process.

A aplicação se cadastra junto ao engenho da entidade da mesma forma que a aplicação Receptor de Comandos, pela ação register NR, só que com as operações *Inform* e *Trap*.

- register NR :: action

```
(10) register NR =
         | get("snmpEngineID")
       then
         | | associate performing-agent with (it, Inform)
         | and
         | | associate performing-agent with (it, Trap)
```

Então, a ação procedure NotificationReceiver é executada eternamente dentro de um *loop*, processando todas as operações de notificação recebidas pela entidade. O funcionamento da ação trap complete é idêntico ao encontrado na ação CommnadResponder-daemon.

8.3 Composição de uma Entidade

Dois exemplos de tipos de netidades serão estudados: entidades com a função de gerente (Figura 3.3) e com a função de agente (Figura 3.4), conforme o padrão estipulado em [HPW99].

Uma entidade com a função de *gerente* contém os seguintes módulos: um Despachante, um Subsistema de Processamento de Mensagens, um Subsistema de Segurança, pelo menos uma aplicação Receptor de Notificações e diversas aplicações usuárias executando as aplicações Gerador de Comandos e Gerador de Notificações. Cada um deles será executado por um agente da semântica de ações, como visto na ação Manager-daemon abaixo:

- Manager-daemon :: action

(11) Manager-daemon =
```
| | subordinate a dispatcher then bind "dispatcher" to it
| and
| | subordinate an application then bind "CR" to it
| and initiate user-applications
| and initiate MPS
| and initiate SS
hence
| | send a message[to the agent bound to "dispatcher"]
| |                [containing closure abstraction of Dispatcher-daemon]
| and
| | send a message[to the agent bound to "CR"][containing closure abstraction of CR-daemon]
| and activate user-applications
| and activate MPS
| and activate SS
```

A forma geral de uma entidade consiste em subordinar todos os agente da semântica de ações necessários, atribuindo-lhes um identificador, e em seguida lhes fornecer a ação que deverão executar. Desta forma, todos os módulos podem referenciar os outros, trocando mensagens.

Uma entidade com a função de *agente* contém os seguintes módulos: um Despachante, um Subsistema de Processamento de Mensagens, um Subsistema de Segurança, um Subsistema de Controle de Acesso, pelo menos uma aplicação Receptor de Comandos e diversas aplicações usuárias executando as aplicações Gerador de Notificações ou *Proxy Forwarder*. Cada um deles será executado por um agente da semântica de ações, como visto na ação Agent-daemon abaixo:

- Agent-daemon :: action

(12) Agent-daemon =
```
| | subordinate a dispatcher then bind "dispatcher" to it
| and
| | subordinate an application then bind "NR" to it
| and initiate user-applications
| and initiate MPS
| and initiate SS
| and initiate ACS
hence
| | send a message[to the agent bound to "dispatcher"]
| |                [containing closure abstraction of Dispatcher-daemon]
| and
| | send a message[to the agent bound to "NR"][containing closure abstraction of NR-daemon]
| and activate user-applications
| and activate MPS
| and activate SS
| and activate ACS
```

As aplicações usuárias variam para cada entidade que seja formada.

As ações dependentes da implementação são explicadas a seguir. A ação initiate user-applications deve contratar um agente da semântica de ações para cada aplicação usuária que exista na entidade. A ação initiate MPS deve contratar um agente da semântica de ações para cada modelo de processamento de mensagens que exista na entidade, a ação initiate SS para cada modelo de segurança, e a ação initiate ACS para cada modelo de controle de acesso. A ação activate user-applications envia a ação a ser executada por cada aplicação usuária que exista na entidade. A ação activate MPS envia a ação a ser executada por cada modelo de processamento de mensagens, a ação activate SS por cada modelo de segurança, e a ação activate ACS por cada modelo de controle de acesso.

(13) initiate user-applications = □
(14) initiate MPS = □
(15) initiate SS = □
(16) initiate ACS = □
(17) activate user-applications = □
(18) activate MPS = □
(19) activate SS = □
(20) activate ACS = □

As ações do tipo initiate serão compostas por uma ação subordinate para captar um agente da semântica de ações do tipo necessário e então uma ação bind para associar um identificador a este agente. As ações do tipo activate serão compostas por uma ação send para enviar a ação que deve ser executada pelo agente antes subordinado pela ação initiate correspondente.

Genericamente, para inserir um novo módulo X na entidade, faz-se necessário construir uma ação initiate X, para subordinar um agente da semântica de ações para representar o módulo, e uma ação activate X para passar a ação a ser executada por este agente.

8.4 Notação Auxiliar

Esta seção contém definições de dados, ações e produtores auxiliares utilizados na especificação de uma entidade SNMP, incluindo abreviaturas e entidades semânticas.

8.4.1 Ações

A ação associate A with D associa uma aplicação executada pelo agente da semântica de ações A com os dados fornecidos pela tupla D, no caso, o identificador do engenho onde será feito o cadastro e o tipo da operação.

- associate _ with _ :: agent, tuple → action

(21) associate A:agent with (C:contextEngineId, O:pduType) = □

A ação initialize MIBs é dependente da implementação e inicializa as MIBs de uma entidade SNMP.

- initialize MIBs :: action

(22) initialize MIBs = □

8.4.2 Produtores

Da forma como a entidade foi especificada, todos os agente da semântica de ações que são por ela ativados se conhecem e podem se referenciar. A forma usada nos capítulos anteriores, pela qual qualquer módulo referenciava o Despachante era através do produtor the Dispatcher of Entity. Este produtor pode agora ser escrito como:

(23) the Dispatcher of Entity = give the agent bound to "dispatcher"

Outros produtores podem ser definidos desta forma, como por exemplo the MPModel associated with _, dependendo apenas que os modelos existentes sejam retornados conforme seu numero definido em [HPW99]. Caso contrário, *nothing* deve ser retornado para que a ação que chamou o produtor termine com falha.

8.5 Considerações Finais

Este capítulo mostrou como uma entidade SNMPv3 é montada usando semântica de ações e como as informações de gerência são nela armazenadas.

CAPÍTULO 9

Conclusão

Este trabalho consiste na especificação formal da terceira versão do *framework* de gerência de redes denominado *Simple Network Management Protocol* (SNMPv3) utilizando o formalismo de *Semântica de Ações*.

O objetivo deste trabalho é melhorar a comunicação entre seres humanos com relação à semântica dos objetos e das entidades SNMP, visando a verificação de implementações já existentes e a geração automática de novas implementações. Este trabalho serve de suporte para futuras especificações formais de objetos de gerência.

O nosso trabalho consistiu na leitura e interpretação dos documentos que definem informalmente o SNMPv3 e na sua definição formal em semântica de ações, incluindo as operações do protocolo, os principais componentes da entidade SNMP (o Despachante e as aplicações padrão), além da composição dos tipos fundamentais de entidades, que são os agentes e gerentes SNMP.

A especificação das operações SNMP contém as ações que correspondem aos procedimentos seguidos quando se gera e envia uma operação de gerência ou quando se processa sua resposta, além da composição de ações que correspondem à instrumentalização das MIBs.

A especificação do Despachante SNMP é formada por ações que correspondem aos procedimentos seguidos pelo Despachante no envio de mensagens de requisição e de resposta para outras entidades, e na recepção de mensagens de requisição e de resposta vindas de outras entidades.

A especificação das aplicações padrão do SNMP contém as ações que correspondem aos procedimentos seguidos pelas aplicações Gerador de Comandos, Receptor de Comandos, Gerador de Notificações e Receptor de Notificações, que são aplicações responsáveis por gerar e enviar ou receber e processar operações SNMP.

A especificação de uma entidade SNMP é formada por ações executadas por agentes da semântica de ações seguindo a função de módulos de uma entidade SNMP, para compor entidades atuando como gerente SNMP (responsáveis por controlar e monitorar objetos de gerência) ou como agente SNMP (responsáveis por manter objetos de gerência). Também mostrou-se a

Classes de Ações	Descrição
generate	Geração de uma instância de um determinado sorte
associate	Associação da identidade de um agente com uma informação
extract	Extração de campos de uma mensagem
ifnot	Indicação de erro quando um teste não é satisfeito
initialize	Alocação de variáveis localmente usadas num módulo do SNMP
cache	Manipulação de variáveis alocadas localmente
register	Cadastramento de uma aplicação em um engenho SNMP
initiate	Subordinação de agentes
activate	Ativação de agentes já subordinados
daemon	Módulo SNMP com função servidora
procedure	Execução do procedimento seguido por algum módulo do SNMP
operation	Geração e processamento das operações SNMP
VarBind	Processamento de uma *VarBind*

Tabela 9.1: Categorias de ações especificadas neste trabalho.

simplicidade para se inserir novos objetos de gerência em uma entidade, bastando especificar como o objeto será atribuído e consultado (ações get e set para um objeto de gerência) e inserir uma célula de memória na ação initialize MIBs, caso necessário.

Em nosso trabalho, ações que tratam de uma mesma característica semântica, que é descrita em vários lugares e de diferentes formas, foram nomeadas de maneira similar, formando *classes de ações*. Por exemplo, todas as ações que geram dados são denominadas de generate X, onde para cada instância diferente de X é escrita uma nova ação. Esta característica mostrou com maior clareza a utilização de um mesmo conceito em diversos pontos da especificação informal, facilitando seu entendimento. A Tabela 9.1 contém uma lista com as principais classes de ações criadas e quais características semânticas elas descrevem.

Além das contribuições já citadas, o nosso trabalho detectou uma lista de características problemáticas da especificação informal, que são relacionadas a seguir. Acreditamos ser esta uma das mais significativas contribuições deste trabalho.

9.1 Problemas Detectados na Especificação Informal do SNMP

As seguintes características problemáticas provêm da especificação informal em [CMRW96b]:

1. Um PDU é gerado para ser enviado numa operação de gerência. "A GetRequest-PDU is

generated and transmitted at the request of a SNMPv2 application." [CMRW96b, página 10].

Problema: Não há indicação de como obter as *VarBinds* para montar o PDU a ser enviado, nas operações de leitura e escrita.

Solução apresentada: Estes dados são fornecidos como informação transitória para a ação generate VarBindList from _, pela aplicação usuária que iniciou a requisição.

2. No processamento da operação Get, se o nome do objeto de gerência procurado não existir, retorna-se o erro *noSuchObject*. Se o nome existir, e a instância do objeto não estiver presente na MIB, retorna-se o erro *noSuchInstance*. "Otherwise, if the variable binding's name does not have an OBJECT IDENTIFIER prefix which exactly matches the OBJECT IDENTIFIER prefix of any (potential) variable accessible by this request, then its value field is set to 'noSuchObject'. Otherwise, the variable binding's value field is set to 'noSuchInstance'." [CMRW96b, página 10].

   Problema: Não está claro quando deve ser retornado o erro *noSuchInstance*. A explicação dada é incompleta e se confunde com o texto que vem a seguir.

   Solução apresentada: A explicação dada acima foi obtida verbalmente através do contato com pessoas que já tinham testado uma implementação funcional do SNMP. Os testes foram corretamente especificados na ação consultVarBind.

3. Diversos testes são efetuados no processamento da operação *Set* antes que uma alteração seja efetuada num objeto de gerência [CMRW96b, páginas 18 e 19].

   Problema: Não está clara a utilidade destes testes. Além disto, um deles conflita com a função do Subsistema de Controle de Acesso, pois testa visão em MIBs. Nos outros casos, não é dito como obter as propriedades que são testadas (ex:"Otherwise, if the variable binding's name specifies a variable which does not exist but can not be created under the present circumstances (even though it could be created under other circumstances), then the value of the Response-PDU's error-status field is set to 'inconsistentName', and the value of its error- index field is set to the index of the failed variable binding."). É difícil testar uma condição desta natureza antes da alteração ser efetuada.

   Solução apresentada: Indicou-se os testes na ação validateVarBind, porém deixando-os dependentes da implementação.

4. Durante o envio de uma notificação, é indicado que as variáveis devem ser extraídas do campo *Objects* dentro da macro de notificação. "If the OBJECTS clause is present in the invocation of the corresponding NOTIFICATION-TYPE macro, then each corresponding

variable, as instantiated by this notification, is copied, in order, to the variable-bindings field." [CMRW96b, página 22].

Problema: Não está indicado se estas variáveis devem ser anexadas aos seus valores nas MIBs locais antes de serem enviadas.

Solução apresentada: Por ser o mais óbvio, as variáveis foram consultadas, pois a função da notificação envolve informar à outra entidade sobre os dados armazenados nas MIBs da entidade de origem.

5. Quando uma notificação é recebida, para cada caso há um processamento distinto a ser realizado. Uma aplicação usuária deverá executá-la. "presents its contents to the appropriate SNMPv2 application;" [CMRW96b, página 22].

   Problema: A forma pela qual uma aplicação é escolhida para processar a resposta de uma notificação não é claramente indicada.

   Solução apresentada: Usou-se a ação the Application associated with *M*:Notification-Macro para escolher uma aplicação para processar a notificação recebida. Esta ação é dependente da implementação. A forma mais simples de proceder sua implementação é através da composição de tabelas associando aplicações com notificações.

As seguintes características problemáticas provêm da especificação informal em [CHPW99]:

1. Quando uma aplicação envia uma requisição, ela recebe um índice *sendPduHandle* do Despachante para coordenar a recepção de sua resposta. "The Dispatcher generates a sendPduHandle to coordinate subsequent processing." [CHPW99, página 8]. "The value of sendPduHandle is used to determine, in an implementation-defined manner, which application is waiting for a response associated with this sendPduHandle." [CHPW99, página 14].

   Problema: Em nenhum local da especificação informal é dito que deve-se fazer a associação da aplicação que enviou uma requisição com o índice *sendPduHandle* gerado pelo Despachante, e nem como é feita.

   Solução a apresentar: A ação associate the cached "sender" with the generated sendPduHandle deve ser inserida na ação procedure Disp send-request B caso a mensagem seja enviada para outra entidade.

2. Quando uma indicação de erro é retornada pelo modelo de processamento de mensagens no envio de uma resposta, esta indicação é retornada para a aplicação que precessou a operação e gerou a resposta. Isto é feito pela ação send back em procedure Disp send-

response B. "If the result is an errorIndication, the errorIndication is returned to the calling application." [CHPW99, página 10].

Problema: Só que segundo a especificação informal, a aplicação que enviou a operação de resposta (Receptor de Comandos ou Receptor de Notificações) não espera por esta indicação de erro.

Solução a apresentar: As aplicações Receptor de Comandos e Receptor de Notificações devem ser adaptadas para esperar e processar este erro (ações procedure CommandResponder e procedure NotificationReceptor).

3. Segundo a especificação informal, o modelo de processamento de mensagens deve armazenar informações relativas ao remetente de uma mensagem de requisição recebida para se poder remeter uma resposta posteriormente. "The cached information for the original request is retrieved via the stateReference, including ... transportDomain and transportAddress." [CHPW99, páginas 27 e 28].

   Problema: Esta função deveria ser realizada pelo Despachante, pois é de sua responsabilidade controlar o fluxo de mensagens entre entidades. Além disto, se o Despachante controlasse estes dados, simplificaria a especificação de novos modelo de processamento de mensagens, pois estes terão trabalho extra ao implementar diversas vezes o mesmo controle de endereços.

   Solução a apresentar: Fazer que o Despachante armazene e detecte o destino das mensagens de resposta. Neste trabalho, fez-se a ação Dispatcher associated with _, que é dependente da implementação, para detectar a entidade a receber uma resposta.

4. O campo *expectResponse* do *sendPdu-SDU* serve para indicar se uma requisição gerada necessita ou não de uma resposta, sendo esta então aguardada.

   Problema: O campo *expectResponse* não tem utilidade no Despachante e nas aplicações padrão, porque para se saber se uma resposta deve ser esperada o PDU é testado, e se seu tipo for *Trap*, não é necessário responder; caso contrário sim. Quanto ao modelo de processamento de mensagens, este pode testar *pduType* e saber quando uma resposta é necessária, afinal é o modelo de processamento de mensagens que armazena informações para se enviar uma resposta. Porém, segundo a especificação informal o modelo de processamento de mensagens não o utiliza também: "The SNMPv3 Message Processing Model does not use the values of expectResponse ..." [CHPW99, página 27].

   Solução a apresentar: O campo pode ser eliminado da primitiva *sendPdu* se não for encontrada nenhuma outra referência para ele em outro modelo da entidade que não foi

especificado neste trabalho.

5. Existem casos onde não há um modelo de processamento de mensagens ativo para processar uma mensagem. "If the messageProcessingModel value does not represent a Message Processing Model known to the Dispatcher, then an errorIndication (implementation-dependent) is returned to the calling application." [CHPW99, página 8]. Existe casos onde não há uma aplicação ativa para processar uma mensagem. "If no application has registered for the combination, then statusInformation is errorIndication plus snmpUnknownPDUHandlers OID value pair." [CHPW99, página 13].

   Problema: A indicação de erro a ser retornada nos casos acima não está clara.

   Solução apresentada: Devolveu-se diferentes códigos de erros para os casos citados (noMPModel e noApplication), a fim de permitir uma maior flexibilidade no controle dos erros.

6. Um SDU é composto por uma sequência de campos.

   Problema: É difícil de entender qual a finalidade de cada um de seus campos.

   Solução apresentada: Agruparam-se pacotes de dados por subsistema dentro dos SDUs para melhorar a visualização e a compreensão de seus campos (*transportData*, *messageData*, *securityData*, *accessData*).

7. Os índices *request-id*, *sendPduHandle* e *stateReference* têm a função de indexar as mensagens que trafegam transmitindo informação de gerência.

   Problema: Não é fácil perceber a similaridade de suas funções, pois seus nomes nem sempre o indicam.

   Solução apresentada: Estes índices foram agrupados num mesmo sorte, denominado *handle*.

As seguintes características problemáticas provêm da especificação informal em [LMS99]:

1. Para enviar uma requisição, as aplicações Gerador de Comandos e Gerador de Notificações utilizam a primitiva *sendPdu* [LMS99, páginas 6 e 15].

   Problema: A maneira de obter os valores para preencher os campos da primitiva a ser chamada não é indicada.

   Solução a apresentar: Estes dados devem ser inseridos pela aplicação usuária que executar Gerador de Comandos ou Gerador de Notificações, através da ação cache, pois serão acessados pelo produtor the cached na ação procedure Aplic send-request.

2. Uma aplicação armazena *sendPduHandle* após enviar uma requisição para poder correlacioná-la com sua resposta. "The command generator should store the sendPduHandle so that it can correlate a response to the original request." [LMS99, página 7].

   Problema: O índice *sendPduHandle* salvo pela aplicação Gerador de Comandos nunca mais é utilizado, pois o Despachante só manda para uma aplicação as respostas destinadas a ela.

   Solução a apresentar: Caso não seja recebida uma indicação de erro, o índice *sendPduHandle* não precisa ser armazenado na ação procedure CommandGenerator e pode ser ignorado.

3. Seguindo a mesma idéia apresentada no item anterior:

   Problema: Como foi dito, a aplicação Gerador de Comandos espera pelo índice *sendPduHandle*. Entretanto, a aplicação Gerador de Notificações não o espera. Só que o Despachante sempre envia este índice, sem distinguir a aplicação receptora.

   Solução a apresentar: Seguindo a idéia apresentada, deve-se mudar a especificação da ação procedure NotificationOriginator a fim de que ela espere uma indicação de erro, conforme ocorre na ação procedure CommandGenerator, ou mudar o envio do índice *sendPduHandle* no Despachante,

4. Quando algumas condições são testadas após a recepção da resposta de uma requisição a fim de validá-la, se as condições falharem a resposta é descartada e um valor de erro deve ser retornado para a aplicação que gerou a requisição. "If the received values of messageProcessingModel, securityModel, securityName, contextEngineID, contextName, and pduVersion are not all equal to the values used in the original request, the response is discarded." [LMS99, página 8].

   Problema: Não há indicação de qual valor de erro deve ser devolvido.

   Solução apresentada: Devolveu-se um valor de erro diferente para cada teste realizado na ação procedure Aplic receive-response.

5. Seguindo a mesma idéia apresentada no item anterior:

   Problema: Estes testes não são efetuados na aplicação Gerador de Notificações para a operação Inform.

   Solução a apresentar: Usar a ação procedure Aplic receive-response na ação procedure NotificationOriginator para uniformizar os testes para todas as operações de gerência

6. Quando a primitiva isAccessAllowed é usada, um modelo de controle de acesso deve ser utilizado para processar a consulta requisitada [LMS99, página 11].

Problema: Não está indicado como o modelo de controle de acesso a ser usado é escolhido.

Solução apresentada: Usou-se a ação ACModel of entity para realizar esta escolha, sendo esta ação dependente da implementação neste estágio da especificação.

7. Quando uma aplicação Receptor de Comandos recebe uma requisição a processar, deve-se testar se a operação recebida é por ela processada ou não. "The operation type should always be one of the types previously registered by the application." [LMS99, página 11].

   Problema: Caso a operação não seja processável pela aplicação Receptor de Comandos, não é indicado o erro que deve ser retornado ou o que deve ser feito.

   Solução apresentada: Retornou-se o valor discardByOperation como indicação de erro na ação procedure Aplic receive-request.

8. Seguindo a mesma idéia apresentada no item anterior:

   Problema: A operação a ser processada não é verificada pela aplicação Receptor de Notificações, somente pela aplicação Receptor de Comandos.

   Solução apresentada: Usou-se a ação procedure Aplic receive-request tanto na ação procedure NotificationReceiver quanto na ação procedure CommandResponder, pois a única diferença nas duas era a ausência deste teste fundamental, já que a aplicação se cadastra junto ao Despachante.

9. Após processar uma requisição, as aplicações Receptor de Comandos ou Receptor de Notificações formam um *returnResponsePdu-SDU* a enviar [LMS99, página 13].

   Problema: Não é claro como obter o valor do campo *statusInformation* deste SDU (ação procedure Aplic send-response).

As seguintes características problemáticas provêm da especificação informal em [HPW99]:

1. Existem informações utilizadas por um módulo que devem ser guardadas durante o processamento de uma operação. Estas informações são armazenadas em uma LCD: "The subsystems, models, and applications within an SNMP entity may need to retain their own sets of configuration information. The collection of these sets of information is referred to as an entity's Local Configuration Datastore (LCD)." [CMRW96b, página 27].

   A seguir encontra-se um exemplo deste tipo de informação: "The request-id is extracted from the PDU and saved." [LMS99, página 11].

   Problema: Não está claro quais informações devem existir na LCD de cada módulo, pois não há uma seção que a descreva inteiramente.

Solução apresentada: Declararam-se somente as informações que foram localmente necessárias (nas ações initialize LCD-Dispatcher, initialize LCD-Generate e initialize LCD-Process).

2. Existem aplicações usuárias definidas em cada entidade. Elas são proprietárias, pois não são padrão.

   Problema: Não há indicação de onde estas aplicações se encaixam na arquitetura de uma entidade (ver exemplos nas Figuras 3.3 e 3.4), que foram extraídas da especificação informal do SNMP.

   Solução apresentada: As aplicações usuárias são inseridas de forma dependente da implementação nas entidades (ver exemplos nas ações Agent-daemon e Manager-daemon).

3. Dentro da composição e uma entidade há aplicações.

   Problema: Identificou-se não ser necessário que as aplicações Gerador de Comandos e Gerador de Notificações sejam executadas como um servidor, por um agente da semântica de ações.

   Solução apresentada: Estas aplicações foram usadas dentro de aplicações usuárias.

As soluções a apresentar não foram realizadas para não alterar a especificação informal contida do SNMP.

9.2 Trabalhos Futuros

O primeiro trabalho a ser realizado é validar a especificação aqui realizada, usando algum método automático.

Outro trabalho é tratar com a questão de tamanho máximo de um pacote [CMRW96b, HPW99] e com a questão de retransmissão de pacotes por tempo, como mencionado na especificação informal da aplicação Gerador de Notificações [LMS99]. Estes aspectos não foram cobertos neste trabalho, podendo se usar descrição de dados como as apresentadas em [Mus92].

Também é necessário especificar outros módulos do engenho e seus modelos usando semântica de ações, incluindo um modelo de processamento de mensagens [CHPW99], um modelo de segurança [BW99], um modelo de controle de acesso [WPM99] e a aplicação padrão *Proxy Forwarder* [LMS99].

Então, finalizando esta série de trabalhos, existe a possibilidade de se gerar código implementável a partir desta especificação formal. Para tanto, é necessário considerar de qual forma serão traduzidas as ações da faceta comunicativa da semântica de ações, pelo fato de que estas

ações ainda não foram traduzidas para código implementável. Uma forma seria interpretar as trocas de mensagens entre agentes como chamadas de procedimentos, que seriam simplificados para a execução de ações em semântica de ações.

BIBLIOGRAFIA

[BMW92] Deryck F. Brown, Hermano Moura, and David A. Watt. ACTRESS: an action semantics directed compiler generator. Technical report, Departmental Research Report FM-1992-1, University of Glasgow, Department of Computing Science, June 1992.

[BW99] V. Blumenthal and B. Wijnen. User-based Security Model (USM) for the SNMPv3. Request for Comments 2574, April 1999.

[CFSD90] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). Request for Comments 1157, May 1990.

[CHPW99] J. Case, D. Harrington, R. Presuhn, and B. Wijnen. Message Processing and Dispatching for the SNMP. Request for Comments 2572, May 1999.

[CMPS99] J. Case, R. Mundy, D. Partain, and B. Stewart. Introduction to Version 3 of the Internet-Standard Network Management Framework. Request for Comments 2570, April 1999.

[CMRW96a] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Management Information Base for Version 2 of the SNMP. Request for Comments 1907, January 1996.

[CMRW96b] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Protocol Operations for Version 2 of the SNMP. Request for Comments 1905, January 1996.

[DM99] Elias P. Duarte Jr. and Martin A. Musicante. Formal Specification of SNMP MIB's Using Action Semantics: The Routing Proxy Case Study. In *Proc. of the Sixth IFIP/IEEE Int'l Symp. on Integrated Network Management*, pages 417–430, Boston, USA, May 1999. IEEE Publishing.

[FMD00a] Digenes C. Furlan, Martin A. Musicante, and Elias P. Duarte Jr. A Formal Description of SNMPv3 Standard Applications using Action Semantics. *Anais do III International Workshop on Action Semantics (AS2000)*, 2000.

[FMD00b] Digenes C. Furlan, Martin A. Musicante, and Elias P. Duarte Jr. An Action

Semantics Description of the SNMPv3 Dispatcher. *Anais do IV Simpsio Brasileiro de Linguagens de Programao (SBLP2000)*, pages 186–199, 2000.

[fS87] International Organization for Standardization. Information Processing Systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1). International Standard 8824, December 1987.

[HPW99] D. Harrington, R. Presuhn, and B. Wijnen. An Architecture for Describing SNMP Management Frameworks. Request for Comments 2571, May 1999.

[LC96] Allan Leinwand and Karen Fang Contoy. *Network Management: a Practical Perspective.* Addison-Wesley, second edition, 1996.

[Lee89] Peter Lee. *Realistic Compiler Generation.* Foundation of Computing Series. The MIT Press, 1989.

[LMS99] D. Levi, P. Meyer, and B. Stewart. SNMPv3 Applications. Request for Comments 2573, May 1999.

[Men98] Luis C. S. Menezes. Uso de orientao a objetos na prototipao de semntica de aes. Master's thesis, Departament of Informatics, Federal University of Pernambuco, 1998.

[Mos89] Peter D. Mosses. Unified algebras and action semantics. In *STACS'89, Proc. Symp. on Theoretical Aspects of Computer Science, Paderborn*, volume 349 of *Lecture Notes in Computer Science.* Springer-Verlag, 1989.

[Mos92] Peter D. Mosses. *Action Semantics.* Cambridge University Press, Cambridge, UK, 1992.

[MPS99a] K. McCloghrie, D. Perkins, and J. Schoenwaelder. Conformance Statements for SMIv2. Request for Comments 2580, April 1999.

[MPS99b] K. McCloghrie, D. Perkins, and J. Schoenwaelder. Structure of Management Information Version 2 (SMIv2). Request for Comments 2578, April 1999.

[MPS99c] K. McCloghrie, D. Perkins, and J. Schoenwaelder. Textual Conventions for SMIv2. Request for Comments 2579, April 1999.

[Mus92] Martin A. Musicante. The Sun RPC language semantics. In *Proceedings of PANEL'92, XVIII Latin-American Conference on Informatics.* Universidad de Las Palmas de Gran Canaria, 1992.

[Mus96] Martin A. Musicante. *On the Relational Semantics of Interleaving Constructors.* PhD thesis, UFPE, Department of Informatics, 1996.

[Ros94] Marshall T. Rose. *The Simple Book: an Introduction to Internet Management.* Prentice Hall, second edition, 1994.

[Sch86] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn & Bacon, 1986.

[sim] The Simpleweb-SNMP / Network Management Software. http://wwwsnmp.cs.utwente.nl/software.

[snm] SNMP Version 3 (SNMPv3). http://www.ibr.cstu-bs.de. Technical University Braunschweig.

[Sta98a] William Stallings. Security Comes to SNMP: The New SNMPv3 Proposed Internet Standards. *The Internet Protocol Journal*, 1(3), 1998.

[Sta98b] William Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2.* Addison-Wesley, third edition, 1998.

[Sta98c] William Stallings. SNMPv3: A Security Enhancement for SNMP. *IEEE Commun. Surveys*, 1998.

[ucd] The UCD-SNMP Project. http://ucd-snmp.ucdavis.edu. University of California, Davis.

[Wat91] David A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall, UK, 1991.

[Win93] Glynn Winskel. *The Formal Semantics od Programming Languages: an Introduction.* Foundations of Computing Series. MIT Press, 1993.

[WPM99] B. Wijnen, R. Presuhn, and K. McCloghrie. View-based Access Control Model (VACM) for the SNMP. Request for Comments 2575, April 1999.

APÊNDICE A

Especificação Completa da Entidade SNMP

A.1 Ações

A.1.1 Entidade

- Manager-daemon :: action

(1) Manager-daemon =

```
| | subordinate a dispatcher then bind "dispatcher" to it
| and
| | subordinate an application then bind "CR" to it
| and initiate user-applications
| and initiate MPS
| and initiate SS
hence
| | send a message[to the agent bound to "dispatcher"]
| |                [containing closure abstraction of Dispatcher-daemon]
| and
| | send a message[to the agent bound to "CR"][containing closure abstraction of CR-daemon]
| and activate user-applications
| and activate MPS
| and activate SS
```

- Agent-daemon :: action

(2) Agent-daemon =

```
    | | subordinate a dispatcher then bind "dispatcher" to it
    | and
    | | subordinate an application then bind "NR" to it
    | and initiate user-applications
    | and initiate MPS
    | and initiate SS
    | and initiate ACS
  hence
    | | send a message[to the agent bound to "dispatcher"]
    | |                [containing closure abstraction of Dispatcher-daemon]
    | and
    | | send a message[to the agent bound to "NR"][containing closure abstraction of NR-daemon]
    | and activate user-applications
    | and activate MPS
    | and activate SS
    | and activate ACS
```

- Dispatcher-daemon :: action

(3)
```
Dispatcher-daemon =
    | initialize LCD-Dispatcher
  hence unfolding
    | | | procedure Dispatcher
    | | trap complete
    | and then unfold
```

- user-application _ :: integer → action

(4)
```
user-application X:integer =
    | initialize LCD-Generate
  hence
    | ...
    | procedure CommandGenerator
    | ...
```

(5)
```
user-application X:integer =
    | initialize LCD-Generate
  hence
    | ...
    | procedure NotificationOriginator
    | ...
```

- CommandResponder-daemon :: action

(6)
```
CommandResponder-daemon =
    | | initialize MIBs
    | and
    | | initialize LCD-Process
    | and
    | | register CR
  hence unfolding
    | | | procedure CommandResponder
    | | trap complete
    | and then unfold
```

- NotificationReceiver-daemon :: action

(7) NotificationReceiver-daemon =
| | initialize LCD-Process
| and
| | register NR
hence unfolding
| | | procedure NotificationReceiver
| | trap complete
| and then unfold

- initialize LCD-Dispatcher :: action

(8) initialize LCD-Dispatcher =
allocate a cell then bind "sender" to it

- initialize LCD-Generate :: action

(9) initialize LCD-Generate =
| rebind
and allocate a cell then bind "transpDom" to it
and allocate a cell then bind "transpAdd" to it
and allocate a cell then bind "MPModel" to it
and allocate a cell then bind "secModel" to it
and allocate a cell then bind "secName" to it
and allocate a cell then bind "secLevel" to it
and allocate a cell then bind "pduVersion" to it
and allocate a cell then bind "contextEngID" to it
and allocate a cell then bind "contextName" to it

- initialize LCD-Process :: action

(10) initialize LCD-Process =
| rebind
and allocate a cell then bind "MPModel" to it
and allocate a cell then bind "secModel" to it
and allocate a cell then bind "secName" to it
and allocate a cell then bind "secLevel" to it
and allocate a cell then bind "pduVersion" to it
and allocate a cell then bind "contextEngID" to it
and allocate a cell then bind "contextName" to it
and allocate a cell then bind "request-id" to it
and allocate a cell then bind "non-repeaters" to it
and allocate a cell then bind "max-repetitions" to it
and allocate a cell then bind "variable-bindings" to it

- initialize MIBs :: action

(11) initialize MIBs = □

- register CR :: action

(12) register CR =

| get("snmpEngineID")
then
| | associate performing-agent with (it, Get)
and
| associate performing-agent with (it, GetNext)
and
| associate performing-agent with (it, GetBulk)
and
| associate performing-agent with (it, Set)

- register NR :: action

(13) register NR =
| get("snmpEngineID")
then
| | associate performing-agent with (it, Inform)
and
| associate performing-agent with (it, Trap)

- associate _ with _ :: agent, tuple → action

(14) associate A:agent with (C:contextEngineId, O:pduType) = □

- initiate user-applications :: action

(15) initiate user-applications = □

- initiate MPS :: action

(16) initiate MPS = □

- initiate SS :: action

(17) initiate SS = □

- initiate ACS :: action

(18) initiate ACS = □

- activate user-applications :: action

(19) activate user-applications = □

- activate MPS :: action

(20) activate MPS = □

- activate SS :: action

(21) activate SS = □

- activate ACS :: action

(22) activate ACS = □

A.1.2 Despachante

- procedure Dispatcher :: action

(23) procedure Dispatcher =
| accept a message[from an agent][containing a SDU]
then
| | procedure Disp send-request
or
| procedure Disp send-response
or
| | procedure Disp receive

- procedure Disp send-request :: action[receiving tuple]

(24) procedure Disp send-request =
| | give (the given tuple)[sendPdu-SDU]
then
| procedure Disp send-request A
then
| send a message[to the MPModel associated with the given messageProcessingModel#3]
| | [containing (the given tuple)[prepareOutgoingMsgIn-SDU]]
and then
| | accept contents of a message[from an MPModel][containing a tuple[prepareOutgoingMsgOut-SD
then
| procedure Disp send-request B
then
| send a message[to the Dispatcher associated with (the given transportDomain#1,
| | the given transportAddress#2)][containing the given Network-MSG#3]

- procedure Disp send-request A :: action[receiving sendPdu-SDU][giving prepareOutgoingMsgIn-SDU]

(25) procedure Disp send-request A =
| | check (the MPModel associated with the given messageProcessingModel#3 is an agent)
or
| | check not (the MPModel associated with the given messageProcessingModel#3 is an agent)
| and then
| | send noMPModel back to sender
| and then
| | escape
and then
| regive and generate SendPduHandle

- procedure Disp send-request B :: action[receiving prepareOutgoingMsgOut-SDU]

(26) procedure Disp send-request B =
| | | check (the given statusInformation#1 is success)
| and then
| | send the generated sendPduHandle back to sender
or
| | check not (the given statusInformation#1 is success)
| and then
| | send the given statusInformation#1 back to sender
| and then
| | escape
and then
| give (the given transportDomain#2, the given transportAddress#3, the given Network-MSG#4)

- procedure Disp send-response :: action[receiving tuple]

(27) procedure Disp send-response =
| | give (the given tuple)[returnResponsePdu-SDU]
then
| | send a message[to the MPModel associated with the given messageProcessingModel#1]
| | [containing (the given tuple)[prepareResponseMsgIn-SDU]]
and then
| | accept contents of a message[from an MPModel][containing a tuple[prepareResponseMsgOut-SD
then
| procedure Disp send-response B
then
| | send a message[to the Dispatcher associated with (the given transportDomain#1,
| | the given transportAddress#2)][containing the given Network-MSG#3]

- procedure Disp send-response B :: action[receiving prepareResponseMsgOut-SDU]

(28) procedure Disp send-response B =

| | check (the given Result#1 is success)
or
| | check not (the given Result#1 is success)
and then
| | send the given Result#1 back to sender
and then
| | escape
and then
| give (the given transportDomain#2, the given transportAddress#3, the given Network-MSG#4)

- procedure Disp receive :: action[receiving tuple]

(29) procedure Disp receive =
| | give (the given tuple)[Network-MSG]
then
| procedure Disp receive A
then
| | send a message[to the MPModel associated with the given messageProcessingModel#1]
| | [containing (the rest of the given tuple)[prepareDataElementsIn-SDU]]
and then
| | accept contents of a message[from an MPModel][containing a tuple[prepareDataElementsOut-SD
then
| | | procedure Disp receive-request B
then
| | | send a message[to the given Application#1]
| | | [containing (the rest of the given tuple)[processPdu-SDU]]
or
| | | procedure Disp receive-response B
then
| | | send a message[to the given Application#1]
| | | [containing (the rest of the given tuple)[processResponsePdu-SDU]]

- procedure Disp receive A :: action[receiving Network-MSG]

(30) procedure Disp receive A =

```
| increment("snmpInPkts")
and then
| | | give the messageProcessingModel extracted from the given Network-MSG
| | trap
| | | | increment("snmpInASNParseErrs")
| | | and then
| | | | escape
| | then
| | | give (the MPModel associated with the given messageProcessingModel)
| | or
| | | | check not (the MPModel associated with the given messageProcessingModel is an agent)
| | | and then
| | | | increment("snmpInBadVersions")
| | | and then
| | | | escape
and then
| give (the transportDomain extracted from the given Network-MSG,
|       the transportAddress extracted from the given Network-MSG)
and then
| regive
```

- procedure Disp receive-request B :: action[receiving prepareDataElementsOut-SDU]

(31) procedure Disp receive-request B =

```
| | | check not (the given Result#1 is sucess)
| | and then escape
| | or
| | | | check (the given Result#1 is sucess)
| | | and then
| | | | check (the given sendPduHandle#11 is none)
| | | and then
| | | | | give the Application associated with (the given contextEngineID#7, the given pduType#10)
| | | trap
| | | | | increment("snmpUnknownPDUHandlers")
| | | | and then
| | | | | | regive and get("snmpUnknownPDUHandlers")
| | | | | then
| | | | | | give (the given messageProcessingModel#2, securityModel#3, securityName#4,
| | | | | |       securityLevel#5, pduVersion#6, contextEngineID#7, contextName#8, PDU#9,
| | | | | |       maxSizeResponseScopedPdu#12, stateReference#14,
| | | | | |       par of(noApplication, the given ObjectValue#15))
| | | | | then
| | | | | | procedure Disp send-response
| | | | and then
| | | | | escape
| | and then
| | give (the given messageProcessingModel#2, securityModel#3, securityName#4, securityLevel#5,
| |       pduVersion#6, contextEngineID#7, contextName#8, PDU#9,
| |       maxSizeResponseScopedPdu#12, stateReference#14)
```

- procedure Disp receive-response B :: action[receiving prepareDataElementsOut-SDU]

(32) procedure Disp receive-response B =

```
| | | check not (the given Result#1 is sucess)
| | and then
| | | escape
| or
| | | check (the given Result#1 is sucess)
| | and then
| | | check not (the given sendPduHandle#11 is none)
| | and then
| | | | give the Application associated with the given sendPduHandle#11
| | | trap
| | | | | increment("snmpUnknownPDUHandlers")
| | | | and then
| | | | | escape
| and then
| give (the given messageProcessingModel#2, securityModel#3, securityName#4, securityLevel#5,
|       pduVersion#6, contextEngineID#7, contextName#8, PDU#9, statusInformation#13,
|       sendPduHandle#11)
```

- generate sendPduHandle :: action[giving sendPduHandle]

(33) generate sendPduHandle = □

A.1.3 Aplicações Padrão

- procedure CommandGenerator :: action

(34) procedure CommandGenerator =

```
| | procedure Aplic send-request
| then
| | send a message[to Dispatcher of entity][containing (the given tuple)[sendPdu-SDU]]
| then
| | accept contents of a message[from Dispatcher of entity][containing a datum]
| then
| | | ifnot (it is a sendPduHandle) generate error it
| | and then
| | | cache it as "sendPduHandle"
| and then
| | | accept contents of a message[from Dispatcher of entity]
| | |                              [containing a tuple[processResponsePdu-SDU]]
| then
| | | procedure Aplic receive-response
```

- procedure CommandResponder :: action

(35) procedure CommandResponder =
| accept contents of a message[from Dispatcher of entity][containing a tuple[processPdu-SDU]]
then
| procedure Aplic receive-request
then
| | give the cached "variable-bindings" then map using abstraction of isAccessAllowed
| and then
| | operationResponse the given opTag#5
then
| procedure Aplic send-response
then
| send a message[to Dispatcher of entity][containing (the given tuple)[returnResponsePdu-SDU]]

- procedure NotificationOriginator :: action

(36) procedure NotificationOriginator =
| | give the given list#2
| then
| | map using abstraction of isAccessAllowed
and then
| | procedure Aplic send-request
| then
| | send a message[to Dispatcher of entity][containing (the given tuple)[sendPdu-SDU]]
and then
| | | check (the given opTag#1 is Inform)
| | and then
| | | | accept contents of a message[from Dispatcher of entity]
| | | | [containing a tuple[processResponsePdu-SDU]]
| | then
| | | give (the error-status of the given Pdu#8, the error-index of the given Pdu#8,
| | | the given variable-bindings of the given Pdu#8)
| or
| | check (the given opTag#1 is Trap)

- procedure NotificationReceiver :: action

(37) procedure NotificationReceiver =
| accept contents of a message[from Dispatcher of entity][containing a tuple[processPdu-SDU]]
then
| procedure Aplic receive-request
then
| | | check (the given opTag#5 is Inform)
| | and then
| | | | operationResponse Inform
| | then
| | | procedure Aplic send-response
| | then
| | | send a message[to Dispatcher of entity][containing (the given tuple)[returnResponsePdu-SDU
| or
| | | check (the given opTag#5 is Trap)
| | and then
| | | operationResponse Trap

- procedure Aplic send-request :: action[giving sendPdu-SDU]

(38) procedure Aplic send-request =

```
| give (the cached "transpDom", the cached "transpAdd", the cached "MPModel",
|       the cached "secModel", the cached "secName", the cached "secLevel",
|       the cached "pduVersion", the cached "contEngID", the cached "contName")
and
| | | check not (the given opTag#1 is GetBulk)
| | and then
| | | operationRequest the given opTag#1 with (0, 0, the given list#2)
| | or
| | | check (the given opTag#1 is GetBulk)
| | and then
| | | operationRequest GetBulk with (the given non-repeaters#3, the given max-repetitions#4,
| | |                                the given list#2)
and
| give not (the given opTag#1 is Trap)
```

- procedure Aplic receive-response :: action[receiving processResponsePdu-SDU]

(39) procedure Aplic receive-response =

```
| | ifnot (the given messageProcessingModel#1 is the cached "MPModel")
| |       generate error discardByMPModel
and
| | ifnot (the given securityModel#2 is the cached "secModel")
| |       generate error discardBySecModel
and
| | ifnot (the given securityName#3 is the cached "secName")
| |       generate error discardBySecName
and
| | ifnot (the given pduVersion#5 is the cached "pduVersion")
| |       generate error discardByPduVersion
and
| | ifnot (the given contextEngineID#6 is the cached "contEngID")
| |       generate error discardByContextEngID
and
| | ifnot (the given contextName#7 is the cached "contName")
| |       generate error discardByContextName
and
| | ifnot (the request-id of the given Pdu#8 is the generated "request-id")
| |       generate error discardByRequestID
and then
| give (error-status of the given Pdu#8, error-index of the given Pdu#8,
|       variable-bindings of the given Pdu#8)
```

- procedure Aplic receive-request :: action[receiving processPdu-SDU][giving (request-id,error-status,Index,list,opTag)]

(40) procedure Aplic receive-request =

```
    | ifnot (the performing-agent is associated with the operation of the given PDU#8)
    |       generate error discardByOperation
    and then
    | | give (request-id of the given PDU#8, error-status of the given PDU#8,
    | |       error-index of the given PDU#8, variable-bindings of the given PDU#8,
    | |       operation of the given PDU#8)
    | then
    | | | regive and cache the splitted PDU
    and then
    | | cache the given messageProcessingModel as "MPModel"
    | and
    | | cache the given securityModel as "secModel"
    | and
    | | cache the given securityName as "secName"
    | and
    | | cache the given securityLevel as "secLevel"
    | and
    | | cache the given pduVersion as "pduVersion"
    | and
    | | cache the given contextEngineID as "contextEngID"
    | and
    | | cache the given contextName as "contextName"
```

- procedure Aplic send-response :: action[giving returnResponsePdu-SDU]

(41) procedure Aplic send-response =

```
    | give (the cached "MPModel",
    |       the cached "secModel", the cached "secName", the cached "secLevel",
    |       the cached "pduVersion", the cached "contEngID", the cached "contName")
    and then
    | give Pdu of (the cached "request-id", the given errorStatus#1,
    |               the given Index#2, the given VarBindList#3) with tag Response
    and then
    | give (size of scopedPdu, the cached "stateReference", the given errorStatus#1)
```

A.1.4 Operações do Protocolo

- operationRequest _ with (_, _, _) :: opTag, Index, Index, tuple → action

(42) operationRequest Op:opTag with (S:Index, I:Index, T:tuple) =

```
    | | generate request-id
    | and
    | | generate VarBindList from T
    then
    | give the PDU of (the given requestId#1, S, I, the given VarBindList#2) with tag Op
```

- generate VarBindList from _ :: tuple → action

(43) generate VarBindList from N:list of ObjectName$^+$ =

```
    | give N
    then
    | map using abstraction of emptyVarBind
```

(44) generate VarBindList from V:VarBindList = regive

(45) generate VarBindList from M:Notification-Macro =

```
| | | get("sysUpTime.0")
| | then
| | | give list of ("sysUpTime.0", the given ObjectValue)
| and
| | | get("snmpTrapOID.0")
| | then
| | | give list of ("snmpTrapOID.0", the given ObjectValue)
| and
| | | generate ObjectNameList from M
| | then
| | | map using abstraction of consultVarBind
| then
| give concatenation(the given list#1, the given list#2, the given list#3)
```

- operationResponse _ :: opTag → action

(46) operationResponse Get =

```
| | setError(noError, 0)
| and
| | | give the cached "variable-bindings"
| | then
| | | map using abstraction of consultVarBind
| trap
| | setError(genErr, the given Index)
| and
| | give the cached "variable-bindings"
```

(47) operationResponse GetNext =

```
| | setError(noError, 0)
| and
| | | give the cached "variable-bindings"
| | then
| | | map using abstraction of consultNextVarBind
| trap
| | setError(genErr, the given Index)
| and
| | give the cached "variable-bindings"
```

```
(48) operationResponse GetBulk =
        | | setError(noError, 0)
        | and
        | | | give min(the cached "non-repeaters", count items of the cached "variable-bindings")
        | | and
        | | | give the cached "variable-bindings"
        | | then
        | | break(the given list#2, the given integer#1)
        | | then
        | | | | give the given list#1
        | | | then
        | | | map using abstraction of consultNextVarBind
        | | and then
        | | | | give the cached "max-repetitions"
        | | | and
        | | | | give the given list#2
        | | | then
        | | | map with repetition using abstraction of consultNextVarBind
        | | then
        | | give concatenation(the given list#1, the given list#2)
        | trap
        | | setError(genErr, the given Index)
        | and
        | | give the cached "variable-bindings"
(49) operationResponse Set =
        | | | give the cached "variable-bindings"
        | | then
        | | map using abstraction of validateVarBind
        | | then
        | | | | setError(noError, 0)
        | | | and
        | | | | map using abstraction of updateVarBind
        | | trap
        | | | setError(commitFailed, the given Index)
        | | and
        | | | | break(the cached "variable-bindings", the given Index)
        | | | then
        | | | | map using abstraction of undoVarBind
        | | trap
        | | | setError(undoFailed, 0)
        | trap
        | | setError(the given errorStatus#1, the given Index#2)
        | then
        | give (the given errorStatus#1, the given Index#2, the cached "variable-bindings")
(50) operationResponse Inform =
```

```
    | setError(noError, 0)
    and
    | give the cached "variable-bindings"
    and
    | | give the Application associated with NotificationTypeId of the cached "variable-bindings"
    | then
    | | send a message [to the given Application][containing the cached "variable-bindings"]
```

(51) operationResponse Trap =

```
    | give the Application associated with NotificationTypeId of the cached "variable-bindings"
    then
    | send a message [to the given Application][containing the cached "variable-bindings"]
```

- generate request-id :: action[giving requestId]

(52) generate request-id = □

- generate ObjectNameList from _ :: Notification-Macro → action[giving ObjectNameList]

(53) generate ObjectNameList from M:Notification-Macro = □

A.1.5 Operações sobre *VarBinds*

- emptyVarBind :: action[receiving VarBind][giving VarBind]

(54) emptyVarBind =

```
    give (the given ObjectName#1, unSpecified)
```

- consultVarBind :: action[receiving VarBind][giving VarBind]

(55) consultVarBind =

```
    | | check (existsObject the given ObjectName#1)
    | and then
    | | | | check not (existsInstance the given ObjectName#1)
    | | | and then
    | | | | | give the given ObjectName#1
    | | | | and
    | | | | | get(the given ObjectName#1)
    | | else
    | | | give (the given ObjectName#1, noSuchInstance)
    else
    | give (the given ObjectName#1, noSuchObject)
```

- consultNextVarBind :: action[receiving VarBind][giving VarBind]

(56) consultNextVarBind =

```
    | | give the next to the given ObjectName#1
    | then
    | | | regive
    | | and
    | | | get(the given ObjectName)
    else
    | give (the given ObjectName#1, endOfMibView)
```

- validateVarBind :: action[receiving VarBind]

(57) validateVarBind =

```
| ifnot (existsObject the given ObjectName#1)
|       generate error notWritable
and then
| ifnot (type(the given VarBind) is type(the given ObjectName#1))
|       generate error wrongType
and then
| ifnot (length(the given VarBind) is length(the given ObjectName#1))
|       generate error wrongLength
and then
| ifnot (encoding(the given VarBind) is encoding(the given ObjectName#1))
|       generate error wrongEncoding
and then
| ifnot (alwaysAtrib(the given ObjectName#1, the given ObjectSyntax#2))
|       generate error wrongValue
and then
| ifnot (alwaysCreate(the given ObjectName#1))
|       generate error noCreation
and then
| ifnot (nowCreate(the given ObjectName#1))
|       generate error inconsistentName
and then
| ifnot (not access(the given ObjectName#1) is READ-ONLY and
|       not access(the given ObjectName#1) is NOT-ACCESSIBLE)
|       generate error notWritable
and then
| ifnot (nowAtrib(the given ObjectName#1, the given ObjectSyntax#2))
|       generate error inconsistentValue
```

- updateVarBind :: action[receiving VarBind][giving VarBind]

(58) updateVarBind =

```
    set(the given ObjectName#1, the given ObjectSyntax#2)
```

- isAccessAllowed :: action

(59) isAccessAllowed =

```
| give (the cached "secModel", the cached "secName", the cached "secLevel",
|       class of the cached "pduType", the cached "contextName", the given ObjectName#1)
then
| send a message[to ACModel of entity][containing then]
then
| accept contents of message[from ACModel of entity][containing an StatusInformation]
then
| | | check all (it is noSuchView, it is noAccessEntry, it is noGroupNome)
| | and then
| | | setError(authorizationError, 0) and give the cached "variable-bidings"
| | | then escape with them
| else
| | | check (it is noSuchContext)
| | and then
| | | setError(0,0) and give empty-list
| | | then escape with them
| else
| | setError(genError,0) and give the cached "variable-bidings"
| | then escape with them
```

A.1.6 Operações sobre Informações de Gerência

- get(_) :: ObjectName → action[giving ObjectValue]

(60) get(N:ObjectName) = □

- set(_,_) :: ObjectName, ObjectSyntax → action

(61) set(N:ObjectName,S:ObjectSyntax) = □

A.1.7 Auxiliar

- map using _ :: abstraction → action[receiving list]

(62) map using A:abstraction =

```
unfolding
| | | check ( the given list is empty-list )
| | and then
| | | give it
| | or
| | | check not ( the given list is empty-list )
| | and then
| | | | | give head of the given list
| | | | | then
| | | | | enact A
| | | | | then
| | | | | give list of the given tuple
| | | | and
| | | | | give tail of the given list
| | | | | then unfold
| | | then
| | | | give concatenation(the given list#1, the given list#2)
```

- map with repetition using _ :: abstraction → action[receiving (natural, list)]

(63) map with repetition using A:abstraction =

```
unfolding
| | | check ( the given natural#1 is 0 )
| | and then
| | | give the empty-list
| or
| | | check not ( the given natural#1 is greater than 0 )
| | and then
| | | | give difference(the given natural#1, 1)
| | | and
| | | | | give the given list#2
| | | | then
| | | | | map using A
| | then
| | | | give the given list#2
| | | and
| | | | unfold
| | | then
| | | | give concatenation(the given list#1, the given list#2)
```

- break _ :: (list, integer) → action[giving (list, list)]

(64) break (L:list, I:integer) =

```
| give 0 and give L
then
| unfolding
| | | | ckeck (the given list#2 is empty-list)
| | | and then
| | | | give (empty-list, empty-list)
| | or
| | | | | ckeck (the given integer#1 is I)
| | | | and then
| | | | | give (empty-list, the given list#2)
| | | or
| | | | | ckeck (the given integer#1 is less than I)
| | | | and then
| | | | | | | give list of head of the given list#2
| | | | | and
| | | | | | | | give sum(the given integer#1, 1)
| | | | | | and
| | | | | | | | give tail of the given list#2
| | | | | then
| | | | | | | unfold
| | | | then
| | | | | | give (concatenation(the given list#1, the given list#2), the given list#3)
```

- accept _ :: message → action[giving tuple]

(65) accept M:message =

```
| receive M
then
| | cache the sender of it as "sender"
| and
| | give the contents of it
```

- accept contents of _ :: message → action[giving tuple]

(66) accept contents of M:message =
    | receive M
    then
    | give the contents of it

- ifnot _ generate error _ :: yielder[of truth-value], errorStatus → action[receiving ObjectName]

(67) ifnot Y:yielder generate error E:errorStatus =
    | check Y
    or
    | | check not Y
    | and then
    | | escape with (E, index of the given ObjectName in the cached "variable-bindings")

- ifnot _ generate error _ :: yielder[of truth-value], errorIndication → action

(68) ifnot Y:yielder generate error E:errorIndication =
    | check Y
    or
    | | check not Y
    | and then
    | | escape with E

- increment(_) :: token → action

(69) increment(N:ObjectName) =
    | get(N)
    then
    | set(N, sum(it, 1))

(70) increment(K:token) =
    store the sum(the integer stored in the cell bound to K, 1) in the cell bound to K

- cache _ as _ :: tuple, token → action

(71) cache D:tuple as K:token =
    store D in the cell bound to K

- cache the splitted PDU :: action

(72) cache the splitted PDU =
    | cache the given requestId#1 as "request-id"
    and
    | cache max(0, the given Index#2) as "non-repeaters"
    and
    | cache the given Index#3 as "max-repetitions"
    and
    | cache the given VarBindList#4 as "variable-bindings"

- setError _ :: (errorStatus, Index) → action[giving (errorStatus, Index)]

(73) setError(S:errorStatus, I:Index) =
    | give (S, I)

- send _ back to sender :: tuple → action

(74) send T:tuple back to sender =
    send a message[to the cached "sender"][containing T]

A.2 Produtores

A.2.1 Entidade

- Dispatcher of entity :: yielder[of agent]

(75) Dispatcher of entity = give the agent bound to "dispatcher"

- ACModel of entity :: yielder[of agent]

(76) ACModel of entity = □

- the generated request-id :: yielder[of requestId]

(77) the generated request-id = □

- the generated sendPduHandle :: yielder[of sendPduHandle]

(78) the generated sendPduHandle = □

- the _ extracted from _ :: tuple, tuple → yielder[of tuple]

(79) the $D \leq$ messageProcessingModel extracted from N:Network-MSG = □

(80) the $D \leq$ transportDomain extracted from N:Network-MSG = □

(81) the $D \leq$ transportAddress extracted from N:Network-MSG = □

- the _ associated with _ :: agent, tuple → yielder[of agent]

(82) the $A \leq$ Application associated with M:Notification-Macro = □

(83) the $A \leq$ Dispatcher associated with N:transportData = □

(84) the $A \leq$ MPModel associated with M:messageProcessingModel = □

(85) the $A \leq$ Application associated with (C:contextEngineID, O:pduType) = □

(86) the $A \leq$ Application associated with H:sendPduHandle = □

- the _ is associated with _ :: agent, opTag → yielder[of truth-value]

(87) the A:agent is associated with O:opTag = □

A.2.2 Operações sobre Informações de Gerência

- existsObject _ :: ObjectName → yielder[of truth-value]

(88) existsObject N:ObjectName = □

- existsInstance _ :: ObjectName → yielder[of truth-value]

(89) existsInstance N:ObjectName = □

- the next to _ :: ObjectName → yielder[of ObjectName]

(90) the next to N:ObjectName = □

- access (_) :: ObjectName → yielder

(91) access(N:ObjectName) = □

- type(_) :: tuple → yielder

(92) type(V:VarBind) = □

(93) type(N:ObjectName) = □

- length(_) :: tuple → yielder[of number]

(94) length(V:VarBind) = □

(95) length(N:ObjectName) = □

- encoding(_) :: tuple → yielder

(96) encoding(V:VarBind) = □

(97) encoding(N:ObjectName) = □

- alwaysCreate(_) :: ObjectName → yielder[of truth-value]

(98) alwaysCreate(N:ObjectName) = □

- nowCreate(_) :: ObjectName → yielder[of truth-value]

(99) nowCreate(N:ObjectName) = □

- alwaysAtrib(_,_) :: ObjectName, ObjectSyntax → yielder[of truth-value]

(100) alwaysAtrib(N:ObjectName, S:ObjectSyntax) = □

- nowAtrib(_) :: ObjectName, ObjectSyntax → yielder[of truth-value]

(101) nowAtrib(N:ObjectName, S:ObjectSyntax) = □

A.2.3 Auxiliar

- the cached _ :: token → yielder[of datum]

(102) the cached K:token =
the datum stored in the cell bound to K

- class of _ :: opTag → yielder

(103) class of 0:opTag = □

- index of _ in _ :: ObjectName, VarBindList → yielder[of Index]

(104) index of N:ObjectName in V:VarBindList = □

- NotificationTypeId of _ :: VarBindList → yielder[of ObjectValue]

(105) NotificationTypeId of V:VarBindList = □

- size of scopedPdu :: yielder[of integer]

(106) size of scopedPdu = □

A.3 Dados

A.3.1 PDU

(107) opTag = Get | GetNext | GetBulk | Set | Inform | Trap | Response | □ (*individual*)

(108) PDU of (R:requestId, S:errorStatus, I:Index, V:VarBindList) with tag T:tag → PDU

(109) request-id of PDU of (R,S,I,V) with tag $T = R$

(110) error-status of PDU of (R,S,I,V) with tag $T = S$

(111) error-index of PDU of (R,S,I,V) with tag $T = I$

(112) variable-bindings of PDU of (R,S,I,V) with tag $T = V$

(113) operation of PDU of (R,S,I,V) with tag $T = T$

- _ tagged with _ :: PDU, tag → PDU

(114) P:PDU tagged with Op:opTag =
if operation of P is Op then P else nothing

A.3.2 SDU

(115) SDU $\geq$ sendPdu-SDU | prepareOutgoingMsgIn-SDU | prepareOutgoingMsgOut-SDU | returnResponsePdu-SDU | prepareResponseMsgIn-SDU | prepareResponseMsgOut-SDU | prepareDataElementsIn-SDU | prepareDataElementsOut-SDU | processPdu-SDU | processResponsePdu-SDU | Network-MSG

(116) sendPdu-SDU = (transportData, messageData, securityData, accessData, expectResponse)

(117) prepareOutgoingMsgIn-SDU = (transportData, messageData, securityData, accessData, expectResponse, sendPduHandle)

(118) prepareOutgoingMsgOut-SDU = (statusInformation, transportData, Network-MSG)

(119) returnResponsePdu-SDU = (messageData, securityData, accessData, maxSizeResponseScopedPdu, stateReference, statusInformation)

(120) prepareResponseMsgIn-SDU = (messageData, securityData, accessData, maxSizeResponseScopedPdu, stateReference, statusInformation)

(121) prepareResponseMsgOut-SDU = (Result, transportData, Network-MSG)

(122) prepareDataElementsIn-SDU = (transportData, Network-MSG)

(123) prepareDataElementsOut-SDU = (Result, messageData, securityData, accessData, pduType, sendPduHandle, maxSizeResponseScopedPdu, statusInformation, stateReference)

(124) processPdu-SDU = (messageData, securityData, accessData, maxSizeResponseScopedPdu, stateReference)

(125) processResponsePdu-SDU = (messageData, securityData, accessData, statusInformation, sendPduHandle)

(126) transportData = (transportDomain, transportAddress)

(127) messageData = (messageProcessingModel)

(128) securityData = (securityModel, securityName, securityLevel)

(129) accessData = (pduVersion, scopedPdu)

A.3.3 MSG

(130) Network-MSG = v3MPMessage | $\square$

A.3.4 Auxiliar PDU

(131) errorStatus = noError | genErr | commitFailed | undoFailed | wrongType | wrongLength | wrongEncoding | wrongValue | noCreation | inconsistentValue | notWritable | inconsistentName | authorizationError (*individual*)

(132) errorStatus $\leq$ Index

(133) Index $\leq$ integer

(134) VarBindList = list of VarBind^{+}

(135) VarBind = (ObjectName, ObjectValue)

(136) ObjectName $\leq$ token

(137) ObjectValue = ObjectSyntax | Exceptions

(138) ObjectSyntax = $\square$

(139) Exceptions = unSpecified | noSuchObject | noSuchInstance | endOfMibView (*individual*)

A.3.5 Auxiliar SDU

(140) sendPduHandle = none — □ (*individual*)

(141) pduType = opTag

(142) statusInformation = Result = success | errorIndication

(143) success = noError

(144) errorIndication = noMPModel | noApplication | discardByMPModel | discardBySecModel | discardBySecName | discardByContextEngID | discardByContextName | discardByPduVersion | discardByRequestID | discardByOperation (*individual*)

(145) par of (E:errorIndication, V:ObjectValue) → errorIndication

A.3.6 Auxiliar

(146) handle ≥ requestId

(147) handle ≥ sendPduHandle

(148) handle ≥ stateReference

(149) agent ≥ Application | MPModel | Dispatcher

(150) Notification-Macro = □

(151) MaxAccess = Not-Acessible | Read-Only | Read-Create | Read-Write | □