CONDADO: Uma Ferramenta para Geração de Testes de Protocolos Combinando CONtrole e DADOs

Selma Bássiga Sabião
selmasab@dcc.unicamp.br

Eliane Martins
eliane@dcc.unicamp.br

Instituto de Computação
Universidade Estadual de Campinas - UNICAMP
Caixa Postal 6176 - CEP: 13083-970 - Campinas, SP - Brasil

Resumo

Este artigo apresenta uma abordagem para geração de casos de testes para protocolos de comunicação a partir de uma Máquina Finita de Estados Estendida (MFEE). Essa abordagem permite tratar os aspectos controle e dados de maneira unificada. Uma ferramenta denominada CONDADO está sendo desenvolvida para dar suporte a esta abordagem. A ferramenta parte da especificação do protocolo feita através de uma linguagem definida nesse trabalho, e transforma essa especificação em um programa em Prolog que será usado para a geração dos casos de testes. Nessa abordagem são utilizadas restrições para o tratamento dos predicados existentes na MFEE e também para limitar o número de casos de testes gerados. Através dessa abordagem, foram gerados casos de testes para a implementação do protocolo de transferência do sistema de comunicação que será usada no sistema de Telecomando do projeto do satélite SACI-1.

Abstract

An approach for generating test cases for communication protocols represented as Extended Finite State Machines (EFSM) is presented in this paper. In this approach, control and data aspects of the protocol are considered. A tool called CONDADO is being developed to support this approach. This tool uses a representation of the protocol in a language defined in this work and transforms this representation into a Prolog program that will be used as test case generator. In this approach, constraints are used to treat predicates and also to reduce the number of test cases. The approach is illustrated on a EFSM of a transfer protocol used in the Telecomand Communication System of the SACI-1 satellite project.

1. Introdução

Nos últimos anos, muitas pesquisas têm sido realizadas na área de teste de protocolos com o objetivo de obter métodos mais sistemáticos e ferramentas automatizadas para testar implementações de protocolos de comunicação.

Testes de protocolos são geralmente do tipo caixa-preta, portanto os métodos existentes para testá-los são baseados na forma como o protocolo é especificado. Para que os testes possam ser gerados automaticamente, é importante que a especificação seja descrita usando algum tipo de formalismo.

Existem diversos formalismos para se especificar protocolos de comunicação, dentre eles: Máquina Finita de Estados (MFE), redes de Petri e gramáticas formais. Três técnicas de descrição formal também foram propostas com esse fim pela ISO e ITU-T (antiga CCITT): SDL, Estelle e Lotos. As duas primeiras são baseadas em extensões de MFE, permitindo com isso tratar não só o aspecto controle (relativo à ordem temporal das interações) quanto o aspecto dados (referente aos parâmetros das interações, valores de variáveis, entre outros).

A mesma especificação, formal ou não, pode levar a diferentes implementações de um protocolo. Por isso é importante a realização dos **testes de conformidade**, os quais têm por objetivo verificar se uma implementação de um protocolo está conforme a sua especificação [Ray87].

O processo de teste é composto por quatro atividades principais: desenvolvimento, especificação, execução e análise dos resultados. Esse trabalho se concentra na primeira atividade do processo de teste, especificamente na geração dos casos de teste, e faz parte de um ambiente denominado ATIFS (Ambiente integrado de Testes baseado em Injeção de Falhas por Software) que engloba todas as atividades de testes acima citadas [Mar95].

Neste artigo é apresentado uma abordagem para geração de testes de protocolo, a partir de uma Máquina Finita de Estados Estendida (MFEE), englobando os aspectos controle e dados. Uma ferramenta denominada CONDADO está sendo desenvolvida para dar suporte a esta abordagem. Esta abordagem foi utilizada para a obtenção de testes para o protocolo de comunicação da camada de transferência da pilha de protocolos que será usada no sistema de Telecomando da estação solo do satélite SACI-1 do INPE [Car97].

Este artigo está estruturado da seguinte forma: inicialmente são apresentados os principais conceitos relacionados a testes de protocolos, seguido da descrição dos trabalhos relacionados. Na seção 3, é apresentado o método proposto para a geração de seqüências de testes, onde são descritos os passos para geração dos casos de testes a partir de um exemplo dado. A seção 4 mostra os resultados obtidos na utilização do método implementado pela ferramenta CONDADO para geração de testes de uma implementação do protocolo para o satélite SACI-1, e finalmente, a seção 5 apresenta as conclusões desse trabalho, bem como, os trabalhos futuros.

2. Trabalhos Relacionados

Muitos estudos sobre a geração de testes para protocolos de comunicação têm sido realizados; em [BP94] é feita uma boa apresentação da área. Nesse artigo apresentaremos somente aqueles que influenciaram na elaboração do método usado nesse trabalho.

Nosso interesse nesse estudo é a geração de testes a partir de Máquinas Finitas de Estados Estendidas (MFEE), as quais estendem as MFE através da inclusão de variáveis de contexto e parâmetros de interações. Além disso, uma transição é caracterizada não só pelo estado origem, estado destino e interação (ou evento) de entrada, como em uma MFE, mas também por predicados e ações. Para que uma transição seja disparada, é necessário a ocorrência do evento de entrada e que o predicado associado seja satisfeito. Ao ser disparada uma transição, a ação associada é executada, a qual geralmente produz uma interação (ou evento) de saída, podendo também afetar as variáveis de entrada. A dificuldade na geração de testes a partir desse tipo de modelo está na interdependência entre o aspecto controle (representado pela MFE) e o aspecto dados (representado pelos predicados e ações).

Em alguns estudos os dois aspectos são tratados separadamente. Por exemplo, no *RNL Conformance Kit* [BKK+89], a especificação é dividida em duas partes: a parte de controle do protocolo é especificada através de uma MFE e a parte de dados é especificada através de ASN.1. Ferramentas de geração de testes utilizam os métodos usuais para os testes de transição de estados, tais como, varredura de transições (método T), seqüência de distinção, entre outros. Para a parte de dados são gerados valores aleatórios para os campos das estruturas descritas em ASN.1. Esses valores são incorporados aos casos de testes gerados pelos métodos de transição de estados.

Outros métodos utilizam grafos de fluxo de controle e grafos de fluxo de dados, utilizados em testes do tipo caixa-branca (conforme [RW85], por exemplo, para uma apresentação sobre o assunto), à especificação, ao invés da implementação. Em [SGC87], é descrita a derivação de testes a partir de Estelle. Neste estudo o grafo de fluxo de controle é representado pela MFE e o grafo de fluxo de dados é criado para representar definição e uso dos parâmetros das primitivas de entrada, das variáveis de contexto e constantes, e das operações de dados (ações). Para tanto a máquina de estados estendida obtida a partir da especificação em Estelle está na forma normalizada, isto é, as ações não contêm instruções de desvio e nem laços.

O método proposto em [Ura87] considera que a especificação do protocolo está na forma normal conforme visto em [SGC87]. A partir dessa especificação normalizada é criado um grafo de fluxo de dados onde são especificadas as associações entre definição e uso de cada variável definida na especificação. Baseado nessa informação, um conjunto de casos de testes é derivado para cobrir todas as definições e uso das variáveis. Essa é uma estratégia dos testes tipo caixa-branca aplicada à especificação. Esse método deve ser usado em complemento aos métodos baseados em fluxo de controle.

Huang também propõe um método baseado em análise do fluxo de dados em [HLJ95]. Esse método cobre os testes de fluxo de dados para protocolos especificados através de MFEE usando uma técnica de análise da executabilidade de transições. Esse método gera todas as seqüências executáveis da MFEE onde são verificadas as definições e uso das variáveis nessas seqüências.

A partir de MFEE, Ramalingom propõe um método que gera casos de testes cobrindo o fluxo de dados e controle do protocolo [RDT95]. Para a parte de fluxo de controle ele propõe uma extensão do método U, que é um dos métodos que usa identificação do estado em testes baseados em transição de estados. Para a parte de dados, ele propõe uma extensão do critério "todos os uso" usado em testes estruturais.

Um outro método baseado em Estelle foi estudado. Esse método, proposto em [MS95], parte da especificação em Estelle e a transforma em uma máquina finita de estados. A partir dessa máquina são gerados casos de teste segundo o método UIO (*Unique Input/Output Sequence*). Nesse método não são realizados testes cobrindo o aspecto de dados do protocolo.

Um ambiente para geração e seleção de testes de protocolos (TESTGEN) também foi importante nesse estudo [VJL+94]. Nesse ambiente, a representação do protocolo é feita a partir de um Sistema de Transição Estendido, que é baseado em Estelle e ASN.1. Nesse sistema são descritos tanto o controle quanto os dados e, a partir dele, são gerados os testes. Uma particularidade interessante desse ambiente é a geração de testes baseados em um mecanismo de restrição, que permite ao usuário especificar quais testes devem ser gerados a partir de um conjunto de entradas, estados e transições. Este mecanismo foi aplicado no método aqui proposto, e será explicado mais adiante.

Uma ferramenta automatizada para geração dos casos de testes também foi importante para esse trabalho. Essa ferramenta, denominada TAG [TPB96], também recebe como entrada uma MFEE, descrita na forma de uma linguagem, e também trata separadamente os aspectos controle e dados. Para o primeiro, utiliza uma variante do método W. No que diz respeito ao aspecto dados, somente parâmetros das interações são levados em conta, os quais devem ser fornecidos pelo usuário.

Castanet propõe um método onde são utilizadas técnicas de testes baseados em transição de

estados à MFEE, a qual é vista como uma versão "compactada" de uma MFE [CS87]. Seu método consiste portanto em "descompactar" ou expandir a MFEE, transformando-a em uma MFE com maior número de estados e de entradas para levar em conta parâmetros de interações e valores de variáveis de contexto presentes nos predicados. Para aqueles predicados para os quais não é possível gerar nem novos estados e nem novas entradas, são usadas heurísticas para a geração dos testes, com base nos ciclos presentes na máquina de estados obtida. Essa solução também foi utilizada nesse trabalho, e será portanto explicada mais adiante.

Outro método estudado e que foi base para esse trabalho é o método de geração de seqüências de teste baseado em gramática [UP83]. Nesse método é construído um gerador de casos de teste em Prolog a partir da especificação do protocolo que pode ser feita através de MFEE. Os autores apresentam um exemplo gerando casos de teste para o aspecto controle. Para o aspecto dados, os autores comentam a possibilidade, mas não mostram nenhuma aplicação.

A ferramenta CONDADO foi criada com base nos trabalhos acima descritos e levando em conta as restrições do ATIFS, onde a especificação do protocolo é feita através de MFEE.

Para construção da ferramenta, foi utilizado, principalmente, o método de geração de teste baseado em gramática [UP83]. Mas, vários outros métodos contribuíram para sua elaboração. Do método proposto em [CS87] surgiu a idéia de identificar os ciclos existentes na MFEE e testar os predicados através dos ciclos. Do TESTGEN [VJL+94], aplicamos a técnica de geração com base nas restrições especificadas pelo usuário, por exemplo, qual transição ou qual entrada deseja-se testar. Na próxima seção iremos descrever esse método para geração de seqüências de testes que está sendo implementado pela CONDADO.

3. Método de Geração de Seqüências de Teste

O método para geração de seqüências de teste implementado pela ferramenta CONDADO é composto de quatro passos [UP83]: (1) inicialmente são levantados os requisitos do protocolo, os quais podem estar especificados em MFEE; (2) esses requisitos são transformados na Linguagem para Especificação do Protocolo (LEP); (3) a partir dessa especificação em LEP, é criado um programa em Prolog, esse programa é chamado de **gerador**; e finalmente (3) o **gerador** é executado para obter as seqüências de teste englobando dados e controle.

A Figura 1 apresenta uma visão geral da ferramenta CONDADO. O passo inicial de especificação do protocolo em LEP é feito pelo usuário. Após essa especificação, a ferramenta executa os próximos passos automaticamente. Nos próximos itens discutiremos cada um dos passos, incluindo a especificação de entrada do protocolo, onde adotaremos MFEE, mas essa especificação de entrada pode ser feita através de um outro formalismo, desde que seja possível sua transformação para LEP.

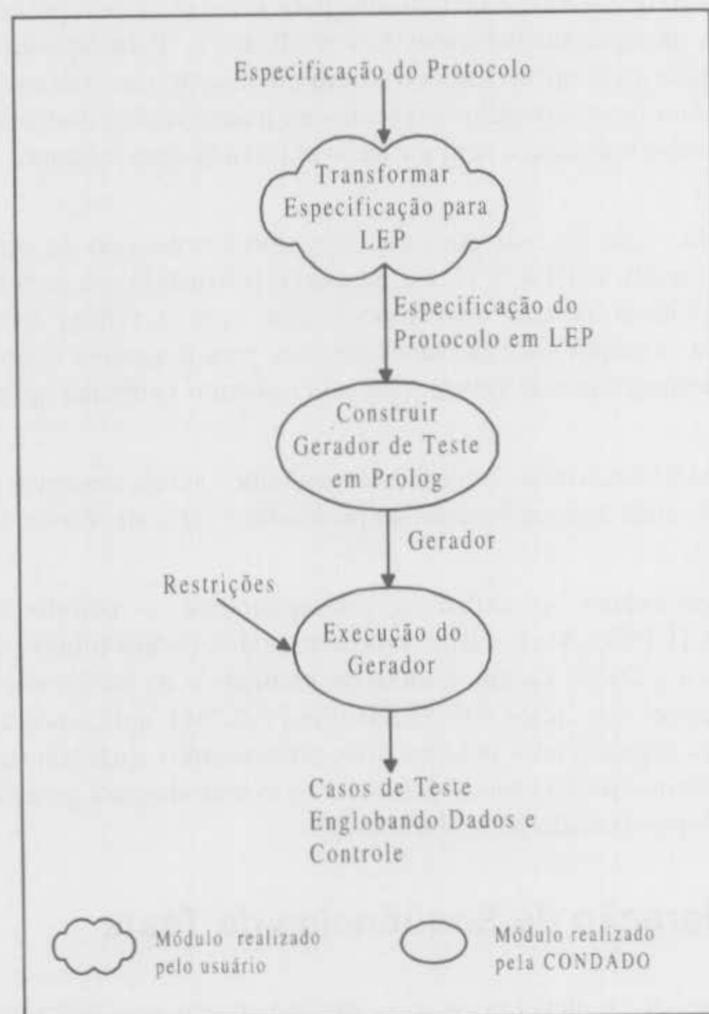

Figura 1 - Visão Geral da Ferramenta CONDADO.

3.1 Especificação do Protocolo na Forma de MFEE

Conforme já foi visto na seção 2, uma MFEE pode ser definida como uma Máquina Finita de Estados (MFE) com variáveis de contexto, predicados e ações. A MFEE deve estar normalizada, ou seja, as ações não podem conter instruções de desvio (*if, CASE)* e nem laços *(for, while, repeat)* [SBC87]. A Figura 2 mostra um exemplo da especificação de um protocolo através de MFEE, que foi retirado de [HLJ95].

Os componentes da MFEE podem ser divididos em um conjunto dos estados e as transições que partem ou chegam a esses estados. Uma transição pode conter entradas, predicados e ações. As entradas são as primitivas de serviço abstratas (ASPs) e unidades de dados de protocolos (PDUs) recebidas pela IUT (implementação em teste). Os predicados são expressões lógicas envolvendo variáveis de contexto e/ou parâmetros das interações; e as ações definem as saídas produzidas e os respectivos parâmetros, podendo também atualizar valores de variáveis.

Cada membro da entrada é representado da seguinte forma: *?U.SENDrequest*, onde *?* representa que é uma entrada e *U* para indicar que esta entrada *SENDrequest* chega à IUT pela camada superior. Cada membro da saída é representado da seguinte forma: *!L.CR*, onde *!* indica que é uma saída e *L* é usado para indicar que a IUT está enviando a saída *CR* para a camada inferior.

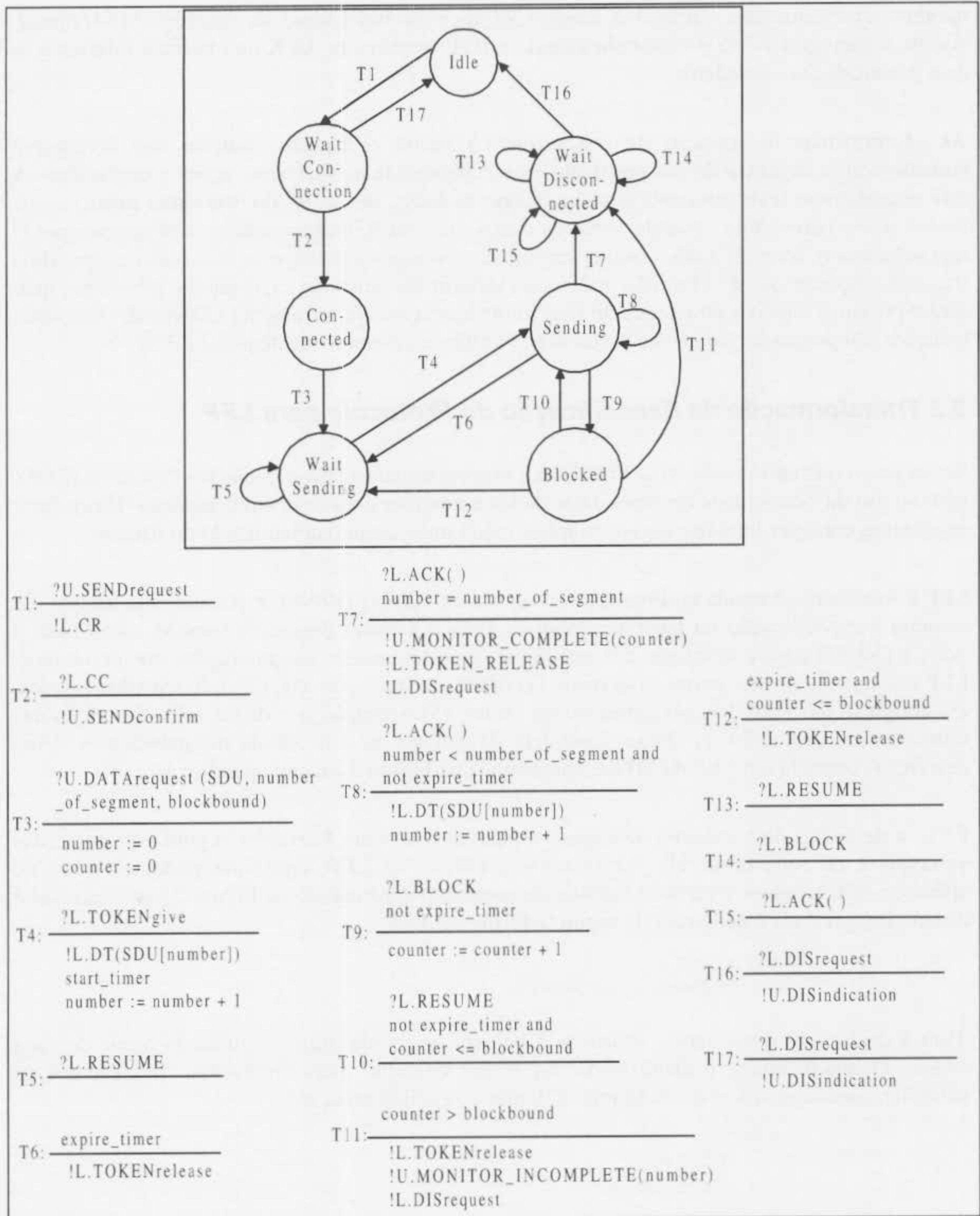

Figura 2: Protocolo especificado através de MFEE.

Uma ação pode modificar o valor de uma variável ou um campo de uma primitiva. Por exemplo, na Figura 2, na transição *T3* é realizada uma ação, onde as variáveis *number* e *counter* são inicializadas. Em *T4* a ação produz uma saída, *DT*, com parâmetro *SDU[number]*, além de inicializar o temporizador e atualizar variável *number*.

Um predicado é utilizado como uma guarda, agindo como controlador das transições da MFEE. Por exemplo, a transição *T7* possui o seguinte predicado: *number = number_of_segment*, onde

number representa uma variável e *number_of_segment* um campo da entrada *DATArequest*. Assim, a transição *T7* só é disparada quando a IUT receber um ACK da interface inferior e se esse predicado for verdadeiro.

As ferramentas de geração de testes que só visam o aspecto controle não levam em consideração o contexto de um protocolo, que compreende as variáveis, ações e predicados. A dificuldade em se levar em conta também o aspecto dados, nos testes do tipo **caixa preta**, é que nesses testes não se tem controle sobre os dados internos à implementação, consequentemente não sabemos o valor de uma variável em um determinado estado, e se a escolha da próxima transição depende desse valor, não poderemos determinar, antes da execução do protocolo, qual será o próximo estado a ser executado pela implementação. Na ferramenta CONDADO algumas soluções são propostas para esses problemas, conforme veremos nos itens 3.3 e 3.4.

3.2 Transformação da Especificação do Protocolo para LEP

Nesse passo o usuário descreve a MFEE na Linguagem para Especificação do Protocolo (LEP). LEP surgiu da necessidade de especificar dados e controle com uma única notação. Outro fator importante era obter uma linguagem simples, facilitando assim o aprendizado do usuário.

LEP é fortemente baseada na linguagem utilizada em TAG [TPB96], especialmente no que diz respeito à especificação da parte de controle. Para o aspecto dados foi tomado como base o ASN.1 [NV92] para a descrição das estruturas de dados usadas nas interações dos protocolos. LEP é composta de seis partes principais: (1) definição das variáveis, (2) definição dos estados, (3) definição das entradas, (4) definição da saídas, (5) definição dos dados e (6) descrição das transições do protocolo. A sintaxe completa da linguagem é mostrada no apêndice A. Uma descrição completa em LEP da MFEE apresentada na Figura 2 está no apêndice B.

Para a definição de variáveis, utiliza-se a palavra reservada *Variables* seguida do nome das variáveis e de seus tipos. Há vários tipos definidos na LEP, conforme pode ser visto no apêndice A. Considere a variável *number* do protocolo apresentado na Figura 2, essa variável é do tipo *integer* e ela é declarada da seguinte forma em LEP:

```
Variables:
        number: integer;
```

Para a declaração dos estados, utiliza-se a palavra reservada *States* seguida do nome de cada estado. O estado inicial é identificado por *#*. Por exemplo, para representar dois estados da máquina, sendo que *idle* é o estado inicial, temos a seguinte notação:

```
States:
        #idle;
        waitconec;
```

As entradas e saídas devem ser especificadas separadamente para facilitar o passo de transformação da LEP para Prolog, conforme será mostrado no item 3.3. As entradas são precedidas pela palavra reservada *Input* e as saídas, pela palavra reservada *Outputs*. Por exemplo:

```
Inputs:
        SENDrequest;
        DISrequest;
        ...

Outputs:
```

```
CR;
DISrequest;
...
```

Em alguns casos, uma interação pode ser tanto de entrada como de saída. Nesse caso ela deve estar discriminada tanto na entrada como na saída. Como por exemplo a interação *DISrequest* que aparece como saída nas transições *T7* e *T11* e como entrada nas transições *T16* e *T17*.

Outro fator importante nas interações de entrada é a diferenciação entre duas transições que são caracterizadas pelo mesmo evento de entrada mas diferem em seus predicados. Quando ocorre uma transição dessa forma e o predicado presente nessa transição envolve valores das interações de entrada, a diferenciação dessa transição é feita acrescentando às interações os valores dos predicados. Esses predicados são chamados de predicados forçáveis [CS87]. Por exemplo, no caso do X.25, um *n_connect_ack.rsp* chamado com parâmetro igual a 1 aceita um *incoming connection* enquanto que com parâmetro zero ele recusa a solicitação de conexão. Para diferenciar essas transições acrescenta-se aos seus parâmetros os valores que eles podem assumir, deixando claro, dessa forma, a transição a ser disparada.

```
n_connect_ack.rsp(RC=1);
n_connect_ack.rsp(RC=0);
```

Para a definição dos dados das primitivas usam-se os tipos criados em ASN.1, quais sejam: *integer, boolean, real, bit string, octet string, enumerated* e *null*. Além desses tipos de dados, pode-se também descrever tipos estruturados, tais como: *sequence, set, sequence of, set of* e *choice*.

Os tipos *sequence* e *set* são similares, ambos são usados para agrupar uma coleção de tipos, incluindo tipos simples e estruturados. A diferença entre eles está na ordenação desses tipos, no tipo *sequence* a ordem em que os tipos são colocados é importante, enquanto que no tipo *set* a ordem não importa. Os tipos *sequence of* e *set of* são similares a um *array* em linguagens de programação, ou seja, eles implicam em uma seqüência de valores que pode ser tanto ordenada (*sequence of*) quanto não ordenada (*set of*). O tipo *choice* permite escolher um, dentre um conjunto de tipos definidos.

Para exemplificar a representação dos dados, considere os tipos de dados pertencentes a estrutura *sequence*, onde o dado *DATArequest* é formado pelos campos *SDU, number_of_segment* e *blockbound*. A descrição dos campos das primitivas começa com a palavra reservada *DATA* seguida da estrutura de dados de cada primitiva.

```
DATA:
DATArequest :: = SEQUENCE {
                          SDU                integer    0..256,
                          number_of_segment  integer    0|2|4,
                          blockbound         integer    0..256 };
```

Na verdade *SDU* é uma unidade de dados da camada de sessão, para simplificar, consideramos como um campo inteiro, mas poderia ser definido com *sequence of*. Para representar valores seqüenciais, como um intervalo de 0 a 256, usa-se a notação *0 .. 256*. Para os valores não seqüenciais, como valores de dados aleatórios, cadeia de caracteres, etc., é necessário discriminá-los um a um, como por exemplo *number_of_segment* que pode ser 0, 2 ou 4.

Após as declarações, são especificadas as transições. A descrição das transições inicia com a palavra reservada *Transitions*, seguida da especificação de cada transição pertencente a máquina. Tomando como exemplo a transição *T1*, sua representação em LEP é feita da seguinte

forma:

```
Transitions:

>idle
?U.SENDrequest
!L.CR
<waitconec;
```

Onde o símbolo > antes do nome do estado indica que *idle* é o estado origem. A entrada é especificada como: *?U.SENDrequest*, onde *?U* indica que o dado *SENDrequest* chegou da camada superior. A saída é representada por *!L.CR* , onde *!L* indica que *CR* será enviado para a camada inferior. Finalmente, o símbolo < indica que o estado *waitconec* é o estado destino da transição.

O exemplo anterior representa uma transição simples, que não possui predicados, ações nem dados relacionados às entradas. Considere agora, exemplos das transições *T3* e *T12* que possuem esses elementos:

```
>connected
?U.DATArequest(SDU, number_of_segment)
{number := 0} {counter := 0}
<waitsend;

>blocked    [expire_timer] [counter <= blockbound]
!L.TOKENrelease
<waitsend;
```

A transição *T3* possui uma ação, na qual as variáveis *number* e *counter* são inicializadas. As ações são representadas entre { }. Nessa transição, como há uma primitiva, seus campos devem ser discriminados, conforme a especificação acima da primitiva *DATArequest*. A transição *T12* possui dois predicados associados, um deles verifica se o tempo de acesso terminou, e o segundo verifica se a variável *counter* é menor ou igual ao dado de entrada *blockbound*. Os predicados são representados entre [ ].

A LEP também permite o uso de comentários, que devem ser precedidos por // no início de cada linha. O fim da especificação é dado pela palavra reservada *END*.

3.3 Construção do Gerador de Teste em Prolog

Com a especificação em LEP, podemos iniciar a execução da ferramenta CONDADO. Esta etapa é automatizada e ela é responsável por transformar a especificação do protocolo feita em LEP num programa em Prolog chamado **gerador**.

A escolha de Prolog se deve ao fato de que a transformação a partir da LEP é quase que direta. Além disso, Prolog oferece o mecanismo de *backtracking* que facilita na geração dos casos de testes por ser um mecanismo que realiza uma busca em largura e profundidade na MFEE. Essa busca é feita com a finalidade de cobrir todos os testes possíveis que possam ser gerados.

Prolog é uma linguagem interpretada composta de *cláusulas de Horn* [CM87]. As cláusulas podem ser *fatos* ou *regras*. Um fato consiste de uma afirmação sobre um objeto. Por exemplo, podemos dizer que "João é do sexo masculino". Trata-se portanto de uma informação sobre o objeto *João*. Em Prolog os fatos são representados da seguinte forma:

sexo(joão, masculino).
progenitor(joão, maria).

Um conjunto de fatos forma uma base de dados (ou base de conhecimento). No fato acima, *joão* está escrito com letra minúscula porque, em Prolog, todo símbolo iniciado por letra maiúscula é uma constante ou variável.

As *regras* são usadas para dizer que um fato depende de um grupo de outros fatos. Ela é formada por uma *cabeça* e um *corpo*. A cabeça da regra aparece antes do símbolo ':-' que é seguido pelo corpo da cláusula. O corpo de uma cláusula pode ser composto de uma ou mais condições, onde a vírgula significa '*e*' e o ponto-e-vírgula significa '*ou*'. Por exemplo:

pai (X,Y) :- progenitor(X, Y), sexo(X, masculino).

Neste caso X e Y são variáveis. Essa regra é interpretada da seguinte forma: *X é pai de Y se X é progenitor e X é do sexo masculino*. A partir desses fatos e regras é possível fazer perguntas para obter informações sobre a base de conhecimento. Por exemplo, com as informações acima podemos perguntar se *joão* é pai de *maria* e Prolog responderá após uma procura em suas cláusulas.

?- pai(joão,maria).
Yes.

Podemos dizer que no **gerador** os valores dos dados são representados por fatos ou regras, sendo que o conjunto de valores dos dados das primitivas representados por fatos formam a base de conhecimento. As transições e alguns dados (como os que possuem valores seqüenciais) são representadas por regras. A construção do gerador será mostrada em detalhes nos itens a seguir.

3.3.1 Determinação dos Ciclos

Há caminhos (seqüência de transições) que podem não ser executáveis devido à existência de predicados incontroláveis [CS87], que representam o efeito do contexto na IUT ou ainda, armazenam os resultados da comunicação entre a IUT e a camada inferior. Estes predicados não podem ser controlados pelos testadores, que representam entidades da camada superior e a entidade par. Um exemplo desse tipo de caminho seria dado pela seguinte seqüência de transições, partindo do estado inicial: T1-T2-T3-T4-T9-T11. Supondo-se que os parâmetros *number_of_segment* e *blockbound* cheguem à T3 com o valor 2, o predicado em T11 seria falso, pois *counter* foi incializado com 0 em T3, e incrementado de 1 em T9, não podendo portanto ser maior do que 2, e assim T11 não pode ser executada. Para que isso aconteça, T9 e T10 devem ser executadas duas vezes. Portanto, para a execução de T11, a seqüência de transições deve ser: T1-T2-T3-T4-T9-T10-T9-T10-T9-T11. Ou seja, o ciclo existente entre os estados *sending* e *blocked* deve ser executado 2 vezes para que a transição T11 possa ser testada, dadas as condições citadas.

O que Castanet e Sijelmassi sugerem, na referência supracitada, é que todos os ciclos da máquina sejam identificados, cabendo ao usuário determinar o número de ocorrências de cada ciclo na seqüência a ser gerada. Tal é o procedimento adotado nesse estudo: a ferramenta determina, antes de iniciar a construção do gerador, quais são os ciclos presentes na máquina e os informa ao usuário.

Em [CS87], os autores definem ainda os predicados controláveis. Esses predicados só dependem de variáveis locais (na Figura 2, *number* e *counter* são variáveis locais). Em [CS87], esses

predicados são eliminados acrescentando novos estados à máquina, caso as variáveis locais tenham valores discretos e limitados. Como essa solução pode ocasionar explosão de estados, no nosso trabalho esses predicados são tratados da mesma forma que os predicados incontroláveis.

Para o exemplo da Figura 2, os seguintes ciclos foram encontrados:

C1: relacionado ao estado *wait sending*.
C2: relacionado ao estado *sending*.
C3: relacionado ao estado *blocked*.
C4: relacionado ao estado *wait disconnected*.

Após a identificação dos ciclos, é iniciada a fase de transformação de LEP para Prolog. Essa fase pode ser dividida em quatro etapas: (1) construir a regra inicial, na qual as variáveis que serão usadas pelas demais regras são inicializadas; (2) construir as regras referentes às transições; (3) construir regras e fatos referentes aos dados; e (4) acrescentar as regras auxiliares para dar suporte às cláusulas de geração dos casos de teste. A seguir falaremos de cada uma dessas etapas exemplificando com o protocolo representado na Figura 2.

3.3.2 Construção da Regra Inicial

Nesse passo, é criada a primeira regra em Prolog, que será o ponto de partida para geração dos casos de testes. Essa regra contém variáveis associadas aos ciclos encontrados em cada estado, além de uma variável que permitirá ao usuário escolher a geração dos testes através de restrições associadas à transição. Essa regra é da seguinte forma:

```
test_sequence(T,C1,C2,C3,C4)  :-    idle(T,[],C1,C2,C3,C4).
```

Onde *T* representa a variável que conterá a restrição do usuário e *C1,C2,C3* e *C4* são os ciclos encontrados na MFEE, mostrados no item anterior. Essa regra leva ao estado *idle* que é o estado inicial da máquina. O símbolo [ ] representa uma lista vazia, que conterá, ao final de cada caminho percorrido pela máquina, os casos de testes gerados.

As restrições são necessárias para restringir o número de sub-percursos considerados para os testes. Casos de testes são gerados para cada sub-percurso iniciando e terminando no estado inicial. O número de sub-percursos em uma MFEE pode ser muito grande, e até infinito, especialmente ao levarmos em conta os ciclos e as variações de valores de parâmetros de interações. O uso de restrições usado aqui está baseado no trabalho de Vuong et Al [VJC+94], implementado na ferramenta TESTGEN. O mecanismo permite ao usuário definir as restrições a serem usadas, oferecendo assim maior flexibilidade a este na geração dos testes.

As seguintes restrições podem ser fornecidas pelo usuário da CONDADO:

- gerar todos os casos de testes para uma determinada transição.
- determinar o número de vezes que cada ciclo pode ser executado.
- gerar casos de testes para todas as transições, passando uma vez em cada ciclo, gerando dessa maneira, todos os casos de testes possíveis para a MFEE, passando no máximo uma vez em cada transição.

3.3.3 Construção das Regras que Representam as Transições

Nessa etapa, todas as transições especificadas em LEP são transformadas em regras. Essa transformação se dá quase diretamente. Observe a transição *T1* representada em LEP:

```
>idle
?U.SENDrequest
!L.CR
<waitconec;
```

Em Prolog, ela ficará da seguinte forma:

```
idle(T,L,C1,C2,C3,C4):- receiveu('SENDrequest',L,L1),
                         transmitl('CR',L1,L2),
                         (((T==[idle,SENDrequest]);(T==[]))->
                               waitconec([],L2,C1,C2,C3,C4);
                               waitconec(T,L2,C1,C2,C3,C4)).
```

Esta regra indica que o estado origem dessa transição é *idle*. Os atributos desse estado são as restrições fornecidas pelo usuário (conforme citado no item anterior), e *L* é a lista contendo as interações que irão compor o caso de teste que está sendo gerado. As cláusulas *receiveu* e *transmitl* são cláusulas auxiliares (ver item 3.3.5) que indicam respectivamente a recepção do evento *SENDrequest* pela camada superior, e transmissão de *CR* para camada inferior.

Abaixo das cláusulas auxiliares tem-se um comando condicional para implementar o mecanismo de restrição usado. Esse comando é interpretado da seguinte forma: *SE a transição atual identificada por [idle,SENDrequest] for igual à transição requerida pelo usuário, cuja identificação está em T, OU T já é vazio ENTÃO é atribuído vazio([ ]) a T indicando que a restrição foi satisfeita. SENÃO o valor de T é passado para a próxima regra.* Em ambos os casos é feita a mudança para o estado *waitconec*. Se ao final das regras, a restrição não foi satisfeita então o caso de teste é desconsiderado.

3.3.4 Construção de Regras e Fatos Para os Dados

Como os dados podem assumir diferentes valores, se faz necessário verificar se o protocolo se comporta corretamente em função das variações desses dados. Como muitas vezes é impossível testar todas as combinações possíveis dos valores dos dados, adotamos uma estratégia de testar aleatoriamente os campos das primitivas e pacotes.

Uma transição que possui uma entrada com dados associados gera uma regra da seguinte forma:

```
connected(T,L,C1,C2,C3,C4):- receiveu('DATArequest',L,L1),
                             data(L1,L2),
                   (((T == ['connected','DATArequest']);(T==[]))->
                              waitsend([],L2,C1,C2,C3,C4);
                              waitsend(T,L2,C1,C2,C3,C4) ).
```

Nessa regra, a diferença está na cláusula `data(L1,L2)` que é responsável por selecionar aleatoriamente os campos da interação de entrada *DATArequest*. Essa cláusula, juntamente com outras auxiliares, devolve valores aleatórios para *SDU, number_of_segment* e *blockbound*.

A estratégia de teste implementada pela CONDADO é a geração aleatória dos valores definidos para os parâmetros das primitivas e pacotes. Outras estratégias podem ser acrescentadas para geração dos dados, tais como, análise de valores-limite e classes de equivalência. Da forma como o programa em Prolog é construído, as estratégias de testes são implementadas através de regras, sendo que o acréscimo de uma nova estratégia é feita pelo acréscimo de novas regras, não sendo necessário, dessa forma, modificar as estratégias já implementadas.

Os dados que possuem valores não seqüenciais são representados através de fatos, onde um fato

é criado para cada valor que o dado pode assumir. Por exemplo, *number_of_segment* pode ter o valor 0, 2 ou 4. Sua representação em Prolog é feita da seguinte forma:

```
number(0,0).
number(1,2).
number(2,4).
```

O primeiro campo do fato representa seu índice, enquanto que o segundo é o valor que *number* pode assumir conforme a especificação. O índice é usado para permitir a escolha aleatória de um conjunto de valores não seqüenciais.

3.3.5 Construção das Cláusulas Auxiliares

Como podemos verificar nas etapas anteriores, várias cláusulas auxiliares são necessárias para gerar os casos de testes. Consideramos como cláusula auxiliar, toda regra ou fato que não especifica um dado ou uma transição diretamente.

Essas cláusulas são fixas e já estão prontas e, portanto, só precisam ser acrescentadas quando um gerador é construído. São responsáveis por funções como: acrescentar um elemento na lista, imprimir uma lista, entre outras funcionalidades necessárias na geração dos testes. As cláusulas auxiliares estão descritas no Apêndice C.

3.4 Casos de Teste Gerados

Após ter construído o gerador de casos de teste, já é possível obter os casos testes a partir da sua execução. Com um conjunto de restrições em mãos, o usuário pode gerar testes significativos para sua implementação de protocolo.

O objetivo dos casos de teste gerados é a cobertura de todas as transições da MFEE e a geração aleatória dos dados das primitivas e pacotes. Um caso de teste é um sub-percurso começando e terminando no estado inicial.

Para mostrar um exemplo dos casos de testes gerados, vamos utilizar a geração seletiva, ou seja, utilizar restrições para delimitar o número de casos de teste gerados. Considerando o gerador do apêndice C, vamos gerar todos os casos de testes com a seguinte restrição: não passar por nenhum ciclo.

```
test_sequence([],0,0,0,0).
 !U.SENDrequest ?L.CR
 !L.CC ?U.SENDconfirm
 !U.DATArequest(223,2,242)
 !L.TOKENgive ?L.DT
 !L.ACK ?U.MONITOR_COMPLETE ?L.TOKEN_RELEASE ?L.DISrequest
 !L.DISrequest ?U.DISindication

 !U.SENDrequest ?L.CR
 !L.DISrequest ?U.DISindication
```

Nesse exemplo, o comando em Prolog `test_sequence([],0,0,0,0)` chama o gerador com a restrição especificada. A notação utilizada para os casos de teste gerados é diferente daquela usada para representação da MFEE. A diferença é que agora estamos olhando do ponto de vista de teste e não da IUT. Nesse caso *!U.SENDrequest* significa que *SENDrequest* será enviado para a IUT pela camada superior *!U*, e *?L.CR* significa que a camada inferior está esperando *CR*

como resposta da IUT à entrada *SENDrequest*.

Como a ferramenta permite o uso de restrições sobre a transição, vamos supor que desejamos testar a transição *T12* passando pelos ciclos no máximo uma vez:

```
test_sequence(['blocked','TOKENrelease'],1,1,1,1).
  !U.SENDrequest ?L.CR
  !L.CC ?U.SENDconfirm
  !U.DATArequest(128,0,32)
  !L.RESUME
  !L.TOKENgive ?L.DT
  !L.ACK ?L.DT
  !L.ACK ?U.MONITOR_COMPLETE ?L.TOKEN_RELEASE ?L.DISrequest
  !L.RESUME
  !L.DISrequest ?U.DISindication
...
```

Para que o usuário possa dizer qual transição ele deseja testar, ele precisa dizer o estado e as entradas e saídas associadas ao estado. Nesse caso, a transição *T12* parte do estado *blocked* e tem como saída *TOKENrelease*. Nós só mostramos um caso de teste gerado, mas para a restrição acima, são gerados 8 casos de teste.

4. Aplicação do Método

Foram gerados casos de testes para uma implementação de um protocolo a fim de validar a abordagem de teste utilizada. O protocolo para o qual foram gerados os testes é um protocolo de comunicação da camada de transferência da pilha de protocolos que será usada no sistema de Telecomando da estação solo do satélite SACI-1 [Car97]. A função deste protocolo é garantir a transmissão de dados de comandos sem erros, omissão ou duplicação, e na seqüência original.

Esse protocolo, representado através de MFEE, possui 6 estados e 46 entradas, e um total de 235 transições. Foram encontrados 9 ciclos nesta MFEE. Devido a grande quantidade de combinações que podem ser obtidas das transições dessa MFEE, foram gerados casos de testes usando restrições. Alguns dos resultados encontrados na geração dos casos de teste são apresentados na tabela 1.

Restrição	Tempo de CPU	Número de Casos de Teste
Não passar pelos ciclos	0.18 segundos	153
Passar 1 vez por 1 ciclo	3.89 segundos	3.213
Passar 1 vez por 4 ciclos	101.98 segundos	70.644
Passar 1 vez por 4 ciclos, cobrindo uma determinada transição.	335.19 segundos	42

Tabela 1 - Dados coletados durante a geração dos casos de teste.

Como podemos verificar na tabela 1, o número de casos de testes diminui consideravelmente quando usamos restrições nas transições. Em compensação o tempo de CPU aumenta devido a forma como o gerador trata as transições (conforme item 3.3.3).

Para o exemplo da Figura 2 apresentado nesse artigo também foram gerados os casos de testes. Para se ter uma idéia da quantidade de testes gerados, na geração dos casos de teste cobrindo todos os caminhos e cobrindo os ciclos no máximo uma vez, foram gerados 57 casos de teste no

tempo de CPU de 0.28 segundos.

5. Conclusão

Nesse artigo apresentamos um estudo sobre os métodos de geração de seqüências de teste para protocolos de comunicação. A partir desse estudo foi derivado um método de geração englobando aspectos de dados e controle do protocolo. Esse método está sendo implementado na ferramenta CONDADO.

Foram descritos os passos de transformação da especificação para obtenção dos casos de teste. A primeira parte do método, a especificação do protocolo na linguagem de especificação definida nesse trabalho, é feita manualmente. Após a obtenção dessa especificação, a ferramenta CONDADO é responsável por gerar um programa em Prolog, chamado gerador, o qual é executado para assim obtermos os casos de teste gerando tanto a parte de controle quanto a parte de dados. Dessa forma, não é necessário tratar separadamente esses aspectos, como acontece na maioria das ferramentas de geração de testes a partir de máquinas finitas de estados estendidas.

A geração dos casos de teste é feita através do uso de restrições, onde o usuário fornece informações como qual transição ele deseja testar ou quantas vezes ele deseja que um determinado ciclo seja executado. Essas restrições foram criadas para auxiliar nos testes dos predicados existentes na especificação e também para limitar o número de casos de testes gerados.

O método proposto foi usado para gerar teste para o protocolo de comunicação da camada de transferência da pilha de protocolos que será usada no sistema de Telecomando da estação solo do satélite SACI-1 do INPE. Com a geração dos casos de teste para esse protocolo, foi possível comprovar a facilidade no uso da LEP, sua transformação para Prolog e também obter medidas do desempenho do gerador em termos de tempo de CPU e do número de casos de teste gerados.

A ferramenta está em fase de implementação e futuramente pretendemos acrescentar outras estratégias de teste para os dados, como teste de valores-limite. Outra extensão para esse trabalho é gerar casos de teste cobrindo as falhas de comunicação, para que possamos verificar o comportamento da implementação em teste na presença de falhas.

Agradecimentos

Agradecemos ao CNPq pelo apoio financeiro para a realização desse trabalho.

6. Bibliografia

[BKK+89] S.P. v. Burgt, J. Kroon, E. Kwast, H.J. Wilts. The RNL Conformance Kit. *2nd Int. Workshop on Protocol Test Systems*, pp. 295-310, Berlim, 3-6 out. 1989.

[BP94] G. v. Bochmam, A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pp. 109-124, ago. 1994.

[Car97] M.J.M. Carvalho. Implementação e Testes do Protocolo da Camada de Transferência do Sistema de Telecomando da Estação Solo do Satélite SACI-1. *Tese de Mestrado*, Universidade Federal do Rio Grande do Norte, Natal, dez. 1997.

[CM87] W.F. Clocksin, C.S.Mellish. Programming in Prolog. 3rd Edition, Springer-Verlag, 1987.

[CS87] R. Castanet, R. Sijelmassi. Methods and Semi-Automatic Tools for Preparing Distributed Testing. *Protocol Specification, Testing, and Verification VI*, pp. 177-187, 1987.

[HLJ95] C.M.Huang, Y.C.Lin, M.Y.Jang. An Executable Protocol Test Sequence Generation Method for EFSM-specified Protocols. *IWPTS - 8th International Workshop on Protocol Test Systems*, pp. 289-305, Evry France, 4-6 set. 1995.

[Mar95] E. Martins. ATIFS: um Ambiente de Testes baseado em Injeção de Falhas por Software. Relátorio Interno DCC-95-24, Departamento de Ciência da Computação, Universidade Estadual de Campinas, dez. 1995.

[MS95] V.B. Mazzola, L. O. R. Silva. Utilização da Técnica Estelle para Gerar Seqüências de Teste para Protocolos de Comunicação. *13o. Simpósio Brasileiro de Redes de Computadores*, pp. 3-22, mai. 1995.

[NV92] G. Neufeld, S. Vuong. An Overview of ASN.1. *Computer Networks and ISDN System*, 23, pp. 393-415, 1992,.

[Ray87] D. Rayner. OSI Conformance Testing. *Computer Networks and ISDN Systems*. 14, pp. 89-98, 1987.

[RDT95] T. Ramalingom, A. Das, K. Thulasiraman. A Unified Test Case Generation Method for the Model Using Context Independent Unique Sequences. *IWPTS - 8th International Workshop on Protocol Test Systems*, pp. 289-305, Evry France, 4-6 set. 1995.

[RW85] S.Rapps, E.J.Weyuker. "Selecting Software Test Data using Data Flow Information". *IEEE Trans. on Software Engineering*, SE-11, n.4, abr. 1985.

[SBC87] B. Sarikaya, G. v. Bochmann, E. Cerny. A Test Design Methodology for Protocol Testing. *IEEE Transactions on Software Engineering*, vol. SE-13, n. 5, pp. 518-531, mai. 1987.

[TPB96] Q. M. Tan, A. Petrenko, G. v. Bochmann. A Test Generation Tool for Specifications in the Form of State Machines. *RL:http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html*. Fev. 1996.

[UP83] H. Ural, R. L. Probert. User-Guided Test Sequence Generation. *Protocol*

Specification, Testing, and Verification III, pp. 421-436, 1983.

[Ura87] H. Ural. Test sequence selection based on static data flow analysis. *Computer Communications*, vol. 10, n. 5, pp. 234-242, out. 1987.

[VJL+94] S. T. Vuong, H. Janssen, Y. Lu, C. Mathieson, B. Do. TESTGEN: An environment for protocol test suite generation and selection. *Computer Communications*, vol. 17, n. 4, pp. 257-270, abr. 1994.

Apêndice A

Gramática Formal para a Linguagem de Especificação da FSM

<FSM_espec> ::= [<variables_defs>] <states_defs> <inputs_defs> <outputs_defs> <data_defs> <transitions_defs> **END** ‘.’
<variables_defs> ::= **VARIABLES** ‘:’ <variable_def> ‘;’ {<variable_def> ‘;’ }
<states_defs> ::= **STATES** ‘:’ <state_def> ‘;’ {<state_def> ‘;’ }
<inputs_defs> ::= **INPUTS** ‘:’ <input_def> ‘;’ {<input_def> ‘;’ }
<outputs_defs> ::= **OUTPUTS** ‘:’ <output_def> ‘;’ {<output_def> ‘;’ }
<data_defs> ::= **DATA** ‘:’ <data_def> ‘;’ {<data_def> ‘;’ }
<transitions_defs> ::= **TRANSITIONS** ‘:’<transition_def>‘;’ {<transition_def> ‘;’ }
<variable_def> ::= <variable_name> ‘:’ <type>
<state_def> ::= [‘#’] <state_name>
<input_def> ::= <input_name>
<output_def> ::= <output_name>
<data_def> ::= <input_name> ‘::=’ <structured_type> ‘{’ (<data_name> <type> <data_value>) { (<data_name> <type> <data_value>) } ‘}’
<transition_def> ::= ‘>’ <state_name> <transition_cont> ‘>’ <state_name>
<transition_cont> ::= <input> [<condition>] [<action>] <output> | [<input>] <condition> [<action>] <output> | <input> [<condition>] [<action>] [<output>]
<structured_type> ::= **SEQUENCE | SET | SEQUENCE OF | SET OF | CHOICE**
<input> ::= ‘?’ (‘U’ | ‘L’) <input_name>
<output> ::= ‘!’ (‘U’ | ‘L’) <output_name>
<condition> ::= ‘[‘ boolean_exp ‘]’
<action> ::= ‘{‘ program_statements ‘}’
<data_value> ::= <value> [‘..’ <value>] ‘,’ | <id> { ‘|’ <id> } ‘,’
// n .. m - significa n, n+1, .. , m-1, m
// n | m | k - são valores não sequenciais cadeias de caracteres
<type> ::= integer | boolean | real | bit string | octet string | enumerated | null
<value> ::= digit {digit}
<id> ::= [‘’’] (letter | digit) {letter | digit} [‘’’]
<variable_name> ::= <name>
<state_name> ::= <name>
<input_name> ::= <name>
<output_name> ::= <name>
<data_name> ::= <name>

<name> ::= letter { letter | digit }

Apêndice B

Especificação do Protocolo em LEP

```
Variables:
      number: integer;
      counter: integer;

States:
      #idle;
      waitconec;
      connected;
      waitsend;
      sending;
      blocked;
      waitdisc;

Inputs:
      SENDrequest;
      CC;
      DATArequest;
      TOKENgive;
      RESUME;
      ACK;
      BLOCK;
      ACK;
      DISrequest;

Outputs:
      CR;
      SENDconfirm;
      DT;
      TOKENrelease;
      MONITOR_COMPLETE;
      TOKEN_RELEASE;
      DISrequest;
      MONITOR_INCOMPLETE;
      DISindication;


DATA:
DATArequest :: = SEQUENCE {
                        SDU                   integer          0..256,
                        number_of_segment     integer          0|2|4,
                        blockbound            integer          0..256 };

Transitions:

>idle        ?U.SENDrequest        !L.CR               <waitconec;
>waitconec   ?L.CC                 !U.SENDconfirm      <connected;
>waitconec   ?L.DISrequest         !U.DISindication    <idle;
>connected   ?U.DATArequest(SDU, number_of_segment)
```

```
            {number := 0} {counter := 0}                          <waitsend;
>waitsend   ?L.RESUME                                             <waitsend;
>waitsend   ?L.TOKENgive {start_timer} {number:=number + 1}
                              !L.DT(SDU[number])                  <sending;
>sending    [expire_timer]    !L.TOKENrelease                     <waitsend;
>sending    ?L.ACK [number = number_of_segment]
                              !U.MONITOR_COMPLETE(counter)
                              !L.TOKEN_RELEASE
                              !L.DISrequest                       <waitdisc;
>sending    ?L.ACK [number<number_of_segment] [not expire_timer]
            {number:=number + 1} !L.DT(SDU[number])               <sending;
>sending    ?L.BLOCK [not expire_timer]
            {counter:=counter + 1}                                <blocked;
>blocked    ?L.RESUME [not expire_timer] [counter <= blockbound]
                                                                  <sending;
>blocked    [counter > blockbound]   !L.TOKENrelease
                                     !U.MONITOR_INCOMPLETE(number)
                                     !L.DISrequest                <waitdisc;
>blocked    [expire_timer] [counter <= blockbound]
                                     !L.TOKENrelease              <waitsend;
>waitdisc   ?L.RESUME                                             <waitdisc;
>waitdisc   ?L.BLOCK                                              <waitdisc;
>waitdisc   ?L.ACK                                                <waitdisc;
>waitdisc   ?L.DISrequest            !U.DISindication             <idle;
END.
```

Apêndice C

Parte do Código em Prolog da Especificação do Protocolo do Apêndice B

```
/*------------------------- Regra Inicial ----------------------*/

test_sequence(T,C1,C2,C3,C4)  :-     idle(T,[],C1,C2,C3,C4).

/*------------------ Transições do protocolo --------------------*/

idle(T,L,C1,C2,C3,C4):- receiveu('SENDrequest',L,L1),
                        transmitl('CR',L1,L2),
                        (((T==['idle','SENDrequest','CR']);(T==[])) ->
                               waitconec([],L2,C1,C2,C3,C4);
                               waitconec(T,L2,C1,C2,C3,C4) ).

waitconec(T,L,C1,C2,C3,C4) :- receivel('CC',L,L1),
                              transmitu('SENDconfirm',L1,L2),
                 (((T==['waitconec','CC','SENDconfirm']);(T==[]))->
                              connected([],L2,C1,C2,C3,C4), fail;
                              connected(T,L2,C1,C2,C3,C4), fail ).

waitconec(T,L,C1,C2,C3,C4):- receivel('DISrequest',L,L1),
                             transmitu('DISindication',L1,L2),
                 (((T==['waitconec','DISrequest','DISindication']);
                    (T==[]))-> imprime(L2), nl, nl;
                               true ).

connected(T,L,C1,C2,C3,C4):- receiveu('DATArequest',L,L1),
                             data(L1,L2),
                      (((T == ['connected','DATArequest']);(T==[]))->
```

```
                                waitsend([],L2,C1,C2,C3,C4);
                                waitsend(T,L2,C1,C2,C3,C4) ).

...
/*----------------------- Dados --------------------------------*/

data(L,L7) :- ap('(',L,L1),
              data1('SDU',L1,L2),
              ap(',',L2,L3),
              data1('number',L3,L4),
              ap(',',L4,L5),
              data1('blockbound',L5,L6),
              ap(')',L6,L7).

data1(D,L,L1) :- devdata(D,N), ap(N,L,L1).

devdata('SDU',N) :- random(256,N).
devdata('number',S) :- random(2,N), number(N,S).
devdata('blockbound',N) :- random(256,N).

/* devolve um numero aleatorio entre 0 e F  */
random(F,X) :- X is random(F).

number(0,0).
number(1,2).
number(2,4).

/*----Cláusulas auxiliares para compor uma lista e imprimí-la ------*/

receiveu(P,Y,K)   :-   ap(' ?U.',Y,W), ap(P,W,K).
transmitu(P,Y,K)  :-   ap(' !U.',Y,W), ap(P,W,K).

receivel(P,Y,K)   :-   ap(' ?L.',Y,W), ap(P,W,K).
transmitl(P,Y,K)  :-   ap(' !L.',Y,W), ap(P,W,K).

ap(X,[],[X])          :- !.
ap(X,[L|[]],[L,X]) :- !.
ap(X,[L|Y],[L|Z])  :- ap(X,Y,Z).

imprime([P|[ ]])  :- ((P==' ?L.') -> nl; true), write(P), !.
imprime([P|Z])    :- ((P==' ?L.') -> nl; true), write(P), imprime(Z).
```