UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

FABRÍCIO GIRARDI ANDREIS

# Estudo Comparativo entre a Síntese de Software Manual e Automática para Protocolos de Comunicação em Sistemas Embarcados

Trabalho de Diplomação.

Prof. Dr. Luigi Carro
Orientador

Porto Alegre, junho de 2011.

# AGRADECIMENTOS

A jornada que se encerra com a entrega deste trabalho foi árdua e gratificante. Tenho certeza de que o apoio que recebi durante esse período foi fundamental para me manter nos trilhos. Por isso, registro aqui os agradecimentos devidos.

Primeiramente, agradeço à minha Família. Obrigado, Fabiane, por estar sempre tão conectada a mim, mesmo com a distância, por acreditar nas minhas palavras, por valorizar a nossa amizade, que transcende o carinho familiar, por me dar suporte nos momentos difíceis, por compartilhar da minha visão de mundo tão particular. Obrigado, mãe, pela enorme dedicação que recebi de ti, pela orientação nos momentos difíceis, pelo porto seguro que significas pra mim, por me amar tão grandiosamente. Obrigado, pai, por ser esse modelo de integridade e perseverança, por valorizar cada instante na minha presença, por me incentivar a buscar o novo.

Agradeço aos amigos que conheci durante meu trabalho no Laboratório de Redes. Lembrarei sempre de nossas elevadas discussões filosóficas. Vocês foram muito importantes para eu compreender e valorizar a academia, além de se tornarem grandes amigos.

Agradeço aos colegas Eduardo, Pietro e Tomás por terem compartilhado das agruras dessa caminhada com entusiasmo. Hoje, eu vejo claramente a importância que vocês tiveram para me manter disposto. Obrigado, também, por insistirem tanto para que eu abrisse a mente e enxergasse o claro e notório, aquele que em minha cabeça parecia mentira.

Agradeço aos meus amigos de Caxias do Sul, por compartilharem de experiências maravilhosas nesses cinco anos e por estarem sempre presentes.

Agradeço à minha namorada, Denise, por aparecer de repente e melhorar tanto a minha vida.

Muito obrigado a todos. Essa conquista também é de vocês.

# SUMÁRIO

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ANSI | American National Standards Institute |
| CPU | Unidade Central de Processamento |
| GCC | *GNU Compiler Collection* |
| LCP | *Link Control Protocol* |
| PPP | *Point-to-Point Protocol* |
| OSI | *Open Systems Interconnection* |
| OXF | *Object Execution Framework* |
| UFRGS | Universidade Federal do Rio Grande do Sul |
| UML | Unified Modeling Language |

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

O presente trabalho busca realizar um estudo comparativo entre a síntese automática e a escrita manual de código-fonte de protocolos de comunicação para sistemas embarcados.

Os sistemas embarcados representam a grande maioria dos processadores vendidos no mundo e têm demandado cada vez mais funcionalidades, sendo grande parte delas implementadas em software. O domínio desses sistemas é impulsionado por fatores como o custo, confiabilidade e tempo de projeto. Além disso, devido à crescente necessidade de troca de informações nos sistemas embarcados, é imprescindível a implementação de protocolos para guiar essa comunicação. Portanto, metodologias que forneçam maior abstração e técnicas para a automatização do processo de síntese de código-fonte são fundamentais.

Neste contexto, utilizando ferramentas comerciais, modelamos o autômato de estados finitos que descreve o funcionamento do protocolo PPP, simulamos um cenário de estabelecimento de conexão entre dois pares e, a partir dos modelos, geramos código-fonte capaz de realizar esse cenário. Depois disso, conduzimos um experimento para cada solução automatizada e para uma solução escrita manualmente. Por fim, analisamos os resultados e traçamos comparações de acordo com os seguintes critérios representativos para o domínio dos sistemas embarcados: consumo de energia, consumo de memória, desempenho, abstração, reúso e tempo de desenvolvimento.

**Palavras-Chave:** síntese automática, protocolo de comunicação, PPP, LCP, sistemas embarcados.

# Comparison between Manual and Automated Software Synthesis for Communication Protocols on Embedded Systems

## ABSTRACT

This work aims to perform a comparative study between automatic synthesis and manual writing of source code of communication protocols for embedded systems.

Embedded systems contain the majority manufactured processors and have been demanding more and more functionalities, which are mostly implemented in software. The domain of these systems is driven by characteristics like cost, reliability and design time. Moreover, the implementation of communication protocols are required to deal with the increasing need of information exchange on embedded systems. Thus, methodologies that provide higher levels of abstraction and techniques to automate the source code synthesis process are of paramount importance.

In this context, with the aid of commercial tools, we modeled the finite-state automaton that describes the functioning of PPP protocol, we simulated a scenario for the link establishment between two pairs and we generated source code from these models. After that, we carried out an experiment for each automated solution and for a handwritten solution. Ultimately, we analyzed the results and we compared the solutions according to the following important metrics for the embedded systems: energy consumption, memory consumption, performance, abstraction, reuse and development time.

**Keywords:** automated synthesis, communication protocol, PPP, LCP, embedded systems.

# 1 INTRODUÇÃO

## 1.1 Contexto

### 1.1.1 Sistemas Embarcados

Sistemas embarcados são sistemas computacionais projetados para a execução de tarefas específicas e compostos por uma combinação adequada de hardware e software. Esses sistemas podem conter diversas unidades de processamento e geralmente possuem requisitos de funcionamento ligados à computação de tempo-real. Ao contrário do que se possa imaginar, os computadores pessoais e servidores correspondem a menos de 2% do total de processadores comercializados no mundo atualmente (TURLEY, 2002). O restante desses processadores está presente nas soluções de sistemas embarcados, os quais, assim como computadores de propósito geral, demandam o desenvolvimento de software. Além disso, devido aos avanços em tecnologias de informação e comunicação, a tendência é de que, no futuro, os domínios de aplicação desses sistemas sejam ampliados, já que cada vez mais produtos devem passar a executar algum tipo de software (EBERT, 2009).

Do ponto de vista de projeto, o software embarcado apresenta restrições e requisitos funcionais e não funcionais. Existem significativas limitações impostas pelo hardware, como o tamanho da memória disponível, o consumo de energia e a capacidade de processamento. Além disso, o ciclo de vida dos produtos embarcados tem se tornado cada vez mais reduzido, fazendo com que o tempo de projeto deva ser encurtado para atender a janelas de chegada ao mercado (*time-to-market*) cada vez menores, atingindo a média de apenas 16 meses (VENTURE DEVELOPMENT CORPORATION, 2010).

Nos últimos anos, a demanda por novas funcionalidades nos produtos embarcados vem crescendo rapidamente. Nesse contexto, o software representa a parte principal do projeto de sistemas embarcados. Essa preferência pelo software pode ser explicada por fatores como o menor custo em relação a uma solução complexa em hardware, a sua flexibilidade e possibilidade de adaptação a mudanças nos requisitos ainda na fase de projeto, o reúso de blocos de projetos anteriores e a facilidade de atualização do sistema. Por outro lado, esse processo tem aumentado a complexidade do próprio software, demandando o emprego de processos e metodologias que lidem com ela, mas eficientes ao ponto de explorarem da melhor maneira possível as características do hardware.

Estatísticas recentes mostram que mais de 40% dos projetos de sistemas embarcados encontram-se atrasados (VENTURE DEVELOPMENT CORPORATION, 2010), o que está relacionado com o aumento de complexidade e a diminuição da janela de chegada ao mercado. Os esforços dos projetistas para lidar com a complexidade do projeto

concentram-se no uso de técnicas e ferramentas baseadas em especificações com maior abstração. Essas medidas estão em consonância com as visões defendidas por autores como (SELIC, 2003) e (GOMMA, 2000), de que essa abordagem é o único meio viável para lidar com a complexidade das novas gerações de sistemas embarcados.

### 1.1.2 Protocolos de Comunicação

Em sistemas computacionais, um protocolo é um conjunto de regras, compartilhado por duas ou mais entidades, que define a sequência de operação e o formato das mensagens a serem trocadas durante uma comunicação. O comportamento de cada entidade pode ser descrito por diversos formalismos, sendo o autômato de estados finitos o modelo mais amplamente aceito (GUNAWAN, 1992).

O autômato de estados finitos é uma abstração matemática do comportamento de um sistema. Esse modelo é composto por um número finito de estados, transições entre esses estados e ações a serem tomadas. As transições entre os estados possuem condições formuladas a partir de dados de entrada e as ações executadas podem ser representadas como dados de saída. Dado um conjunto de entradas e o estado atual, o autômato calcula um novo estado e o conjunto de saídas.

O processo de tradução da especificação formal do protocolo, apresentada no parágrafo anterior, para uma representação em software é chamada de implementação. Quando conduzida manualmente, a implementação do protocolo tende a ser tediosa e suscetível a erros (ABDULLAH, 2003). Além disso, essa abordagem manual dificulta a execução de tarefa de simulação da solução, atividade mandatória para a garantia do correto funcionamento do protocolo.

## 1.2 Motivação

Diversos sistemas embarcados atuais (*e.g.*, equipamentos de telecomunicação, telefones celulares, sistemas automotivos) realizam algum tipo de troca de informações. Para tal, grande parte deles implementa protocolos de comunicação em nível de software. Neste sentido, temos um cenário com dois aspectos principais: a diminuição da janela de chegada ao mercado dos produtos embarcados e as dificuldades inerentes à implementação manual de protocolos. Para enfrentar esses problemas, uma das abordagens possíveis é a utilização de ferramentas de síntese automática de software.

Na síntese automática de software, são utilizados modelos formais ou semi-formais para construir uma visão particular da especificação da aplicação, permitindo que detalhes não relevantes em uma determinada etapa do projeto sejam abstraídos. O código-fonte é gerado automaticamente a partir desses modelos, não sendo mais escrito pelo desenvolvedor. Uma das implicações da síntese é que o processo de modelagem deve ser menos custoso do que a escrita manual do código-fonte. O objetivo principal dessa técnica é o aumento da produtividade dos desenvolvedores por meio do deslocamento da atuação desses profissionais para níveis mais altos de abstração. Além disso, os modelos de alto nível podem beneficiar o desenvolvedor nas tarefas de verificação e simulação antes mesmo da geração do código-fonte. Esse é um ponto extremamente positivo, pois, dentro do ciclo de vida de um projeto, quanto mais tarde os problemas forem encontrados, maiores serão os seus custos de correção (SOMMERVILLE, 2006).

## 1.3 Objetivos do Trabalho

O objetivo deste trabalho é a realização de um estudo comparativo entre a implementação manual e automática de um protocolo de comunicação para sistemas embarcados. A realização deste estudo contempla todas as seguintes etapas:

(i) Modelagem de um autômato de estados finitos de um protocolo de comunicação em duas ferramentas de síntese automática de código-fonte.

(ii) Simulação e geração de código-fonte em cada ferramenta.

(iii) Compilação dos códigos-fonte gerados e de um código-fonte escrito manualmente sem otimizações.

(iv) Realização de experimento que exercite as transições do autômato em um ambiente controlado, isto é, onde seja possível computar métricas físicas do sistema embarcado durante a execução da aplicação.

(v) Análise dos resultados do experimento de acordo com critérios relevantes no domínio dos sistemas embarcados.

Antes da execução de cada uma dessas etapas, foi necessário escolher o protocolo de comunicação que serviria de estudo de caso. Para isso, como premissas fundamentais, ele deveria estar presente em sistemas embarcados e ser implementado em nível de software. Por outro lado, nem todo protocolo possui uma especificação formal de seu funcionamento, ficando a cargo de cada desenvolvedor traduzir essa descrição em uma implementação. Isso poderia gerar implementações consideravelmente diferentes entre si, inviabilizando as comparações a que nos propomos. Assim, procuramos por um protocolo cuja especificação contivesse uma descrição formal do comportamento de suas entidades no formato de um autômato de estados finitos, e não apenas recomendações em linguagem natural. Com isso, garantimos que tanto a escrita manual quanto a síntese automática de software devem gerar um código-fonte com a mesma semântica.

Por isso, escolhemos o protocolo *Point-to-Point Protocol* (PPP) para servir de estudo de caso, pois ele atende a todas as restrições mencionadas no parágrafo anterior. Os objetivos e o funcionamento do PPP serão discutidos no restante do trabalho.

## 1.4 Organização do Trabalho

No Capítulo 2, apresentaremos o protocolo PPP, descreveremos o seu funcionamento e ilustraremos um exemplo de sua utilização. No Capítulo 3, detalharemos a implementação do protocolo. Primeiramente, abordaremos o processo de síntese automática desde a modelagem até a geração de código-fonte. Além disso, apresentaremos o código-fonte escrito manualmente, o qual servirá de base para a comparação. O Capítulo 4 realiza uma análise comparativa da qualidade de cada uma das soluções no domínio dos sistemas embarcados. Por fim, o Capítulo 5 apresenta as conclusões sobre os resultados obtidos neste trabalho.

# 2 PROTOCOLO PPP

## 2.1 Introdução

Neste capítulo, apresentaremos o PPP e discutiremos brevemente o seu funcionamento geral. Depois disso, descreveremos o protocolo LCP, parte integrante do PPP, e detalharemos o seu mecanismo de controle, representado por um autômato de estados finitos. Por fim, mostraremos um exemplo de troca de mensagens para o estabelecimento de uma conexão PPP.

## 2.2 Funcionamento

O protocolo PPP (SIMPSON, 1994) define um método para a comunicação de equipamentos sobre enlaces seriais ponto-a-ponto. Considerando o modelo *Open Systems Interconnection* (OSI) para sistemas de comunicação (ZIMMERMANN, 1980), o PPP pertence à camada de enlace e dá suporte ao transporte de datagramas de diversos protocolos da camada de rede. Ele foi projetado para prover uma solução fácil e comum para a conexão de uma grande variedade de dispositivos e é usado sobre diversos tipos de conexões físicas, tais como cabo serial, linha telefônica, enlaces de rádio ou de fibra ótica.

Para estabelecer a comunicação no enlace ponto-a-ponto, cada par envia, primeiramente, informações para configurar e testar a transmissão de dados. Depois que a conexão estiver estabelecida, os pares podem ou não ser autenticados. A seguir, são escolhidos e configurados um ou mais protocolos da camada de rede, o que permite o envio de seus datagramas pelo enlace. A conexão permanece configurada para comunicações até que um dos pares ou o administrador do sistema decida encerrá-la. Durante todo esse processo, o enlace controlado pelo PPP passa por fases distintas, sendo que cada uma delas é controlada por um protocolo específico. Esses protocolos são partes integrantes da especificação do PPP. A Figura 2.1 ilustra um diagrama dessas fases e o protocolo utilizado em cada uma. Já a Tabela 2.1 descreve os processos de cada fase.

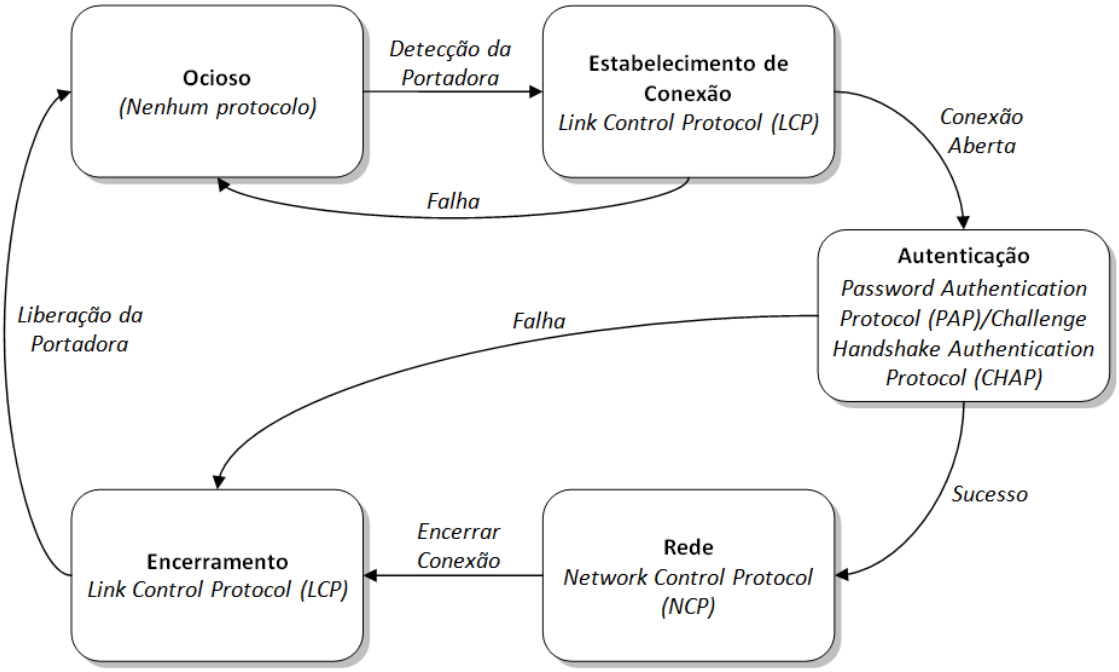


Figura 2.1: Diagrama contendo as fases do protocolo PPP.

Tabela 2.1: Funcionamento de cada fase do protocolo PPP.

| *Fase* | *Descrição* |
|---|---|
| Ocioso | Indica que o meio físico ainda não está pronto para ser usado. No momento em que um evento externo indica que a camada física está pronta para ser usada, tal como a detecção de uma portadora, o PPP passa para a fase de estabelecimento de conexão. |
| Estabelecimento de Conexão | São negociadas opções de configuração da camada de enlace, ou seja, aquelas que independem dos protocolos utilizados na camada de rede. Exemplos de opções negociadas são: o tamanho máximo dos quadros, o tipo de protocolo de autenticação, compressão dos dados, entre outros. |
| Autenticação | Os pares autenticam uns aos outros antes de iniciarem a troca de pacotes da camada de rede. Essa fase é opcional. |
| Rede | Cada protocolo de rede é configurado separadamente. Depois disso, mas ainda nessa fase, a troca de pacotes da camada de rede pode ser executada. |
| Encerramento | Compreende tanto o encerramento gracioso como aquele decorrente de uma falha (*e.g.*, perda de portadora, falha de autenticação). |

## 2.3 Protocolo LCP

Dentre os protocolos membros da especificação do PPP, o *Link Control Protocol* (LCP) foi utilizado como objeto de estudo deste trabalho por ser descrito através de uma especificação formal, conforme a discussão da Seção 1.3. O LCP é utilizado para negociar opções de formato de encapsulamento, controlar limites variáveis no tamanho

dos pacotes, detectar enlaces em *loop* e outras configurações errôneas, além de encerrar a conexão do enlace.

Para descrever o funcionamento do mecanismo de controle do LCP, o formalismo utilizado é um autômato de estados finitos, descrito na Subseção 1.1.2. No autômato que define o LCP, as entradas são representadas por eventos, os quais têm seus significados descritos na Tabela 2.2 e podem ser de três tipos: (i) recepção de comandos externos (*e.g.*, abertura ou fechamento da conexão); (ii) expiração de um temporizador; ou (iii) recebimento de pacotes de um dos pares.

Tabela 2.2: Significados dos eventos do autômato de estados finitos do LCP.

| *Nome* | *Significado do Evento* |
|---|---|
| *Up* | Camada inferior está pronta para transportar pacotes. |
| *Down* | Camada inferior não está mais pronta para transportar pacotes. |
| *Open* | Um agente externo indica que o enlace está disponível para o tráfego, ou seja, apto a ser aberto. |
| *Close* | Um agente externo indica que o enlace não está disponível para o tráfego. |
| TO+ | O temporizador para chegada de resposta expirou e o pacote deve ser reenviado. |
| TO- | O temporizador para chegada de resposta expirou e não há reenvio. |
| RCR+ | Um pacote de Requisição de Configuração foi recebido e as opções de configuração foram aceitas pelo par. |
| RCR- | Um pacote de Requisição de Configuração foi recebido e as opções de configuração foram rejeitadas pelo par. |
| RCA | Uma Confirmação de Requisição de Configuração foi recebida, indicando que as opções de configuração foram aceitas pelo par. |
| RCN | Uma Rejeição de Requisição de Configuração foi recebida, indicando que as opções de configuração foram rejeitadas pelo par. |
| RTR | Uma Requisição de Encerramento de Conexão foi recebida. |
| RTA | Uma Confirmação de Encerramento de Conexão foi recebida. |
| RUC | Um pacote não interpretável foi recebido. |
| RXJ+ | Um aviso de Rejeição de Protocolo ou de Código foi recebido, mas a conexão continua válida. |
| RXJ- | Um aviso de Rejeição de Protocolo ou de Código foi recebido e a conexão é encerrada. |
| RXR | Um pacote de verificação do status da conexão foi recebido. |

As saídas do autômato são representadas por ações, as quais são descritas na Tabela 2.3 e também podem ser de três tipos: (i) reinicialização de um temporizador ou contador; (ii) invocação de comandos externos; ou (iii) transmissão de pacotes para um dos pares.

A Tabela 2.4 descreve os estados e atribui um número de identificação a eles.

Tabela 2.3: Descrição das ações do autômato de estados finitos do LCP.

| *Ação* | *Descrição* |
|---|---|
| tlu | Indica às camadas superiores que o autômato está entrando no estado *Opened*. |
| tld | Indica às camadas superiores que o autômato está saindo do estado *Opened*. |
| tls | Indica à camada inferior a entrada no estado de Inicialização. |
| tlf | Indica à camada inferior que ela não é mais necessária à conexão. |
| irc | Reinicializa o contador de reenvios a um configurável. |
| zrc | Reinicializa o contador de reenvios para zero |
| scr | Envio de um pacote de Requisição de Configuração |
| sca | Envio de um pacote de Confirmação de Requisição de Configuração. |
| scn | Envio de um pacote de Rejeição de Requisição de Configuração. |
| str | Envio de um pacote de Requisição de Encerramento de Conexão. |
| sta | Envio de um pacote de Confirmação de Encerramento de Conexão. |
| scj | Envio de um pacote de Rejeição de Código, indicando o recebimento de um pacote desconhecido. |
| ser | Envio de um pacote de verificação do status da conexão. |

Tabela 2.4: Descrição dos estados do autômato de estados finitos do LCP.

| *Número* | *Nome* | *Descrição* |
|---|---|---|
| 0 | *Initial* | Camada inferior está indisponível. |
| 1 | *Starting* | Um pacote indicando a abertura (*Open*) da conexão foi recebido, mas a camada inferior continua indisponível. |
| 2 | *Closed* | O enlace está disponível (*Up*), mas fechado. |
| 3 | *Stopped* | O enlace está disponível (*Up*) e aberto (*Open*), mas ainda não foram negociadas as opções de configuração. |
| 4 | *Closing* | O fechamento do enlace foi requisitado e uma tentativa de encerramento de conexão foi enviada, mas sua confirmação ainda não foi recebida. |
| 5 | *Stopping* | Uma tentativa de encerramento de conexão foi enviada e sua confirmação ainda não foi recebida. |
| 6 | *ReqSent* | Uma Requisição de Configuração foi enviada. |
| 7 | *AckRcvd* | Uma Confirmação de Requisição foi recebida |
| 8 | *AckSent* | Uma Confirmação de Requisição foi enviada |
| 9 | *Opened* | A conexão está aberta. |

A Figura 2.2 ilustra o diagrama de estados do autômato. As transições de estados possuem legendas no formato “condições : ações”, sendo múltiplas condições separadas pelo operador “ou” e múltiplas ações separadas por vírgulas. Para tornar o diagrama mais compacto, cada seta pode conter múltiplas linhas de pares condições/ações, ou seja, ela pode representar mais de uma transição. Os eventos não especificados em condições não causam transição no estado em questão.

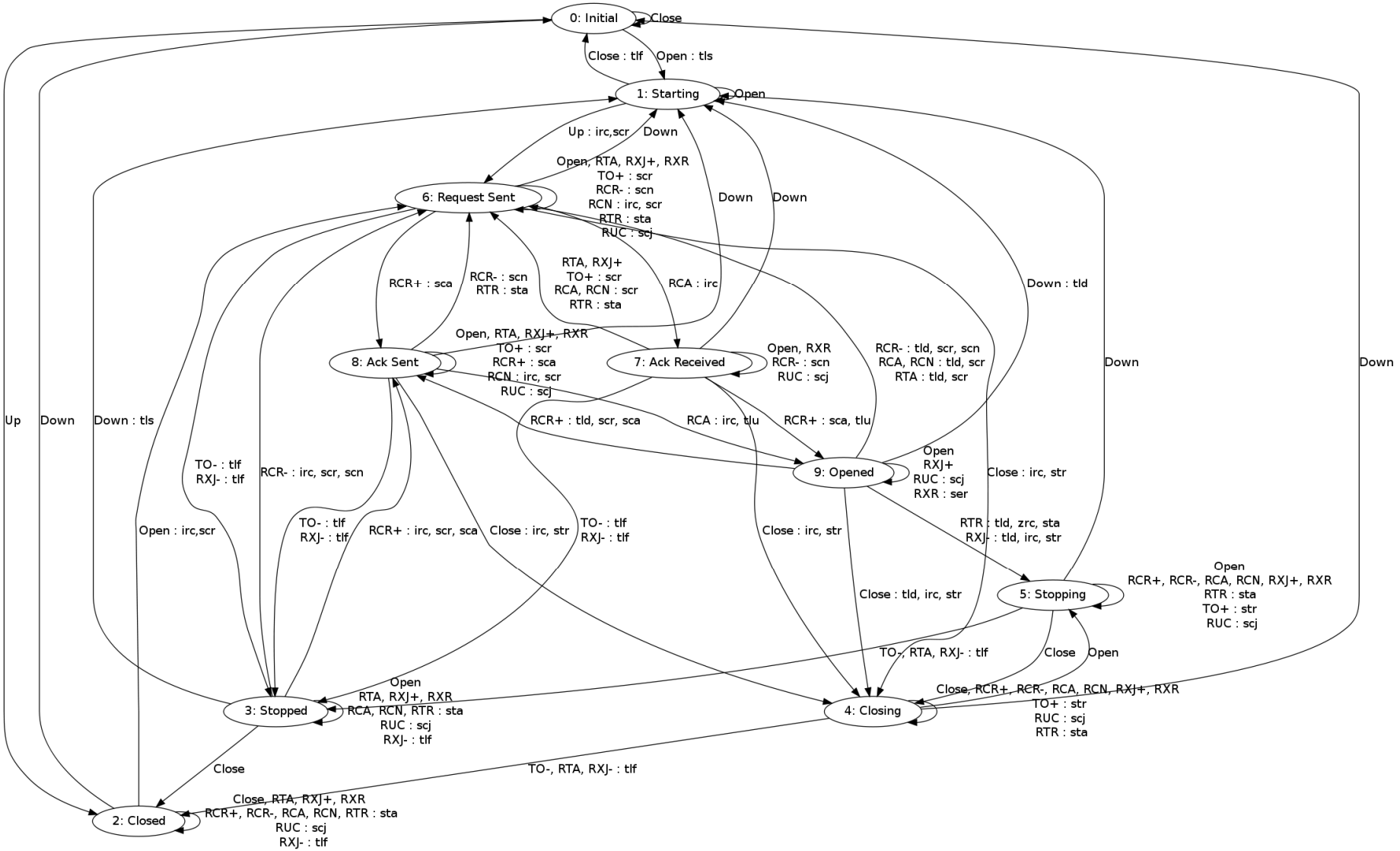


Figura 2.2: Diagrama de estados do autômato de controle do protocolo LCP.

Ambos os pares do enlace ponto-a-ponto seguem o comportamento descrito pelo autômato da Figura 2.2. A conexão é estabelecida quando ambos enviarem e receberem confirmações de requisição de configuração, ou seja, quando o estado *Opened* for alcançado. A entrada do autômato no estado *Opened* também representa a transição no diagrama de fases da Fase de Estabelecimento de Conexão para a Fase de Autenticação. Além do estabelecimento, o LCP também será utilizado para o término da conexão durante a Fase de Encerramento.

## 2.4 Exemplo de estabelecimento de conexão

A Figura 2.3 mostra um possível cenário de configuração e encerramento da conexão PPP sobre o enlace. As trocas de mensagens entre os pares, por meio de pacotes, são representadas por setas sólidas, enquanto que as ações externas realizadas por outros agentes ou expirações de temporizadores são ilustradas por setas com linhas tracejadas. Cada par, ao receber um comando externo ou um pacote, trata essa ocorrência e gera um evento correspondente para o autômato de controle. Os eventos são representados em negrito sobre o ponto de chegada da seta de uma ação. Os novos estados alcançados em resposta a um evento de entrada são representados pelo número correspondente da Tabela 2.4 dentro de um círculo imediatamente abaixo da seta.

Ambos os pares começam no estado *Initial* e a primeira ação executada é *up*, indicando que a camada inferior está pronta para transportar pacotes, o que tipicamente é um sinal do dispositivo de que o meio físico está funcional. Logo em seguida, o administrador do sistema realiza a operação *open* para manifestar a intenção de abrir a conexão. Assim, ambos os pares enviam pacotes de Requisição de Configuração (*ReqConf*), dando início ao processo de negociação. O par Servidor tem suas opções aceitas imediatamente por Modem, recebendo uma confirmação (*ConfConfig*). Já o par Modem, tem suas configurações inicialmente negadas por Servidor (*RejReq*), o que torna necessário o reenvio da requisição e o posterior recebimento da confirmação. Quando ambos atingem o estado *Opened*, a conexão está configurada e ativa, como mencionado anteriormente, e a atividade no enlace passa para as camadas superiores.

O protocolo LCP volta a atuar quando o comando externo (*close*) requisitando o fechamento da conexão é executado pelo administrador do sistema. Modem detecta o evento *Close* e envia uma Requisição de Encerramento de Conexão (*ReqEnc*) a Servidor, que confirma a operação imediatamente (*ConfEnc*) e reinicializa seu temporizador para apenas uma contagem. No momento em que o temporizador expira sua contagem, o par Servidor automaticamente vai para o estado *Stopped*, indicando que a conexão foi interrompida.

De posse da especificação formal do protocolo LCP e de um exemplo de sua utilização para o estabelecimento de conexão, podemos iniciar o processo de modelagem, simulação e geração de código-fonte, como será exposto no capítulo seguinte.

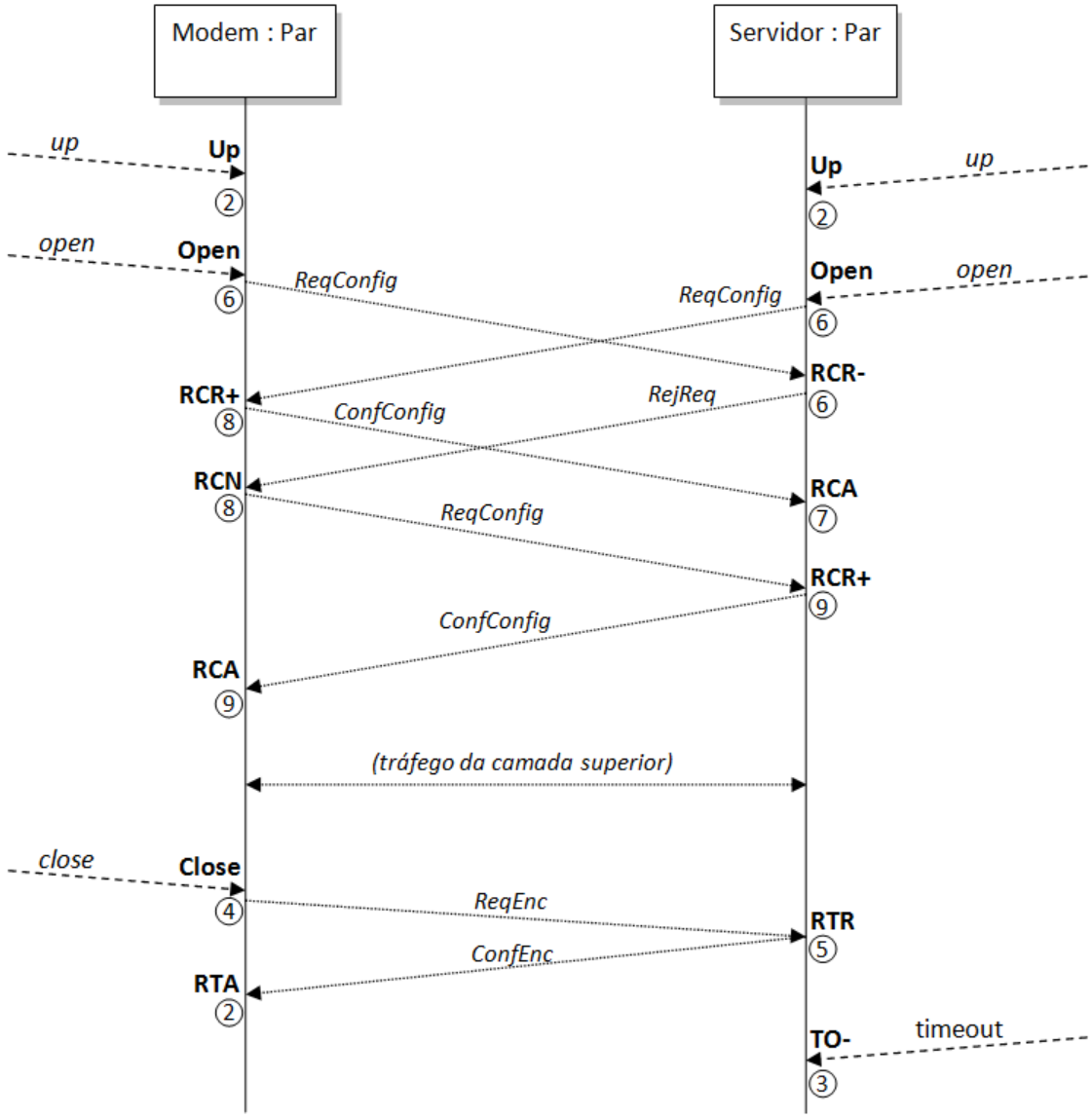


Figura 2.3: Exemplo de configuração e encerramento de uma conexão PPP.

# 3 IMPLEMENTAÇÕES

## 3.1 Introdução

Neste capítulo, detalharemos as implementações do controlador do protocolo LCP, começando pelas duas soluções de síntese automática e finalizando com a solução de escrita manual. Como visto na Seção 1.2, a síntese automática implica na transformação de modelos de alto nível de abstração em código-fonte, um processo que requer a utilização de ferramentas capazes de executar essa tradução. Para o estudo de caso deste trabalho, escolhemos as ferramentas comerciais Simulink (THE MATHWORKS, 2011-b) e IBM Rational Rhapsody (IBM, 2011-b). Optamos por esses softwares devido à sua grande aceitação no mercado e as suas características de modelagem, como será abordado nas seções seguintes.

No restante deste capítulo, apresentaremos as ferramentas comerciais mencionadas no parágrafo anterior e abordaremos suas principais funcionalidades e seus modelos internos de representação de autômatos de estados finitos. Além disso, para cada ferramenta, explicaremos o processo de geração automática e apresentaremos um trecho do código-fonte obtido. Por fim, discutiremos o código-fonte escrito manualmente.

## 3.2 Simulink

### 3.2.1 Descrição da Ferramenta

A ferramenta Simulink (THE MATHWORKS, 2011-b) apresenta um ambiente gráfico baseado em modelos para projeto e simulação de sistemas embarcados e dinâmicos. Ele provê um ambiente interativo e um conjunto personalizável de bibliotecas de blocos que permitem o projeto, simulação, implementação e teste de diversos sistemas, como os de comunicação, processamento de sinais, processamento de vídeo e processamento de imagens. Diversas extensões podem ser inseridas no Simulink com dois objetivos distintos: (i) ampliar a capacidade de modelagem para múltiplos domínios; e (ii) disponibilizar novas ferramentas para a realização de tarefas específicas, como a geração de código-fonte.

Neste trabalho, são utilizadas duas dessas extensões: o Stateflow e o Real-Time Workshop. O Stateflow (THE MATHWORKS, 2011-c) disponibiliza um ambiente para o desenvolvimento de máquinas de estados e diagramas de fluxo, permitindo a representação gráfica dos estados e das transições entre eles. Além disso, é possível detectar conflitos e inconsistências do modelo e simular a sua execução. Já o Real-Time Workshop (THE MATHWORKS, 2011-a) gera e executa código-fonte C e C++ a partir de diagramas do Stateflow.

### 3.2.2 Modelagem e Simulação

O formalismo para representação de autômatos de estados finitos no Stateflow segue a especificação de *Statecharts* apresentada em (HAREL, 1987). *Statechart* é uma extensão do modelo convencional do diagrama de transição com essencialmente três elementos extras, os quais tratam, respectivamente, das noções de hierarquia, paralelismo e comunicação. Com essas diferenças, o diagrama de transição torna-se mais estruturado, compacto e modular, ou seja, a sua facilidade de interpretação é aumentada ao mesmo tempo em que ele se mantém equivalente a um autômato de estados finitos.

Para representar o autômato do protocolo LCP, descrito na Seção 2.3, na ferramenta Simulink utilizando o formalismo de Statecharts, foi necessário definir de que maneira os eventos de entrada e as ações de saída seriam descritos. Como o objetivo principal do trabalho é realizar uma avaliação do funcionamento interno do controlador do protocolo, representado pelo autômato, decidimos construir uma interface simples. Nessa interface, os eventos são sinais recebidos em portas de entrada individuais e as ações são sinais enviados para portas de saída individuais, sendo que os dados das portas são booleanos. A Figura 3.1 ilustra essa construção. O bloco central denominado FSM é o modelo do autômato na ferramenta Stateflow, as setas numeradas à esquerda são as portas de entrada e as setas numeradas à direita são as portas de saída.

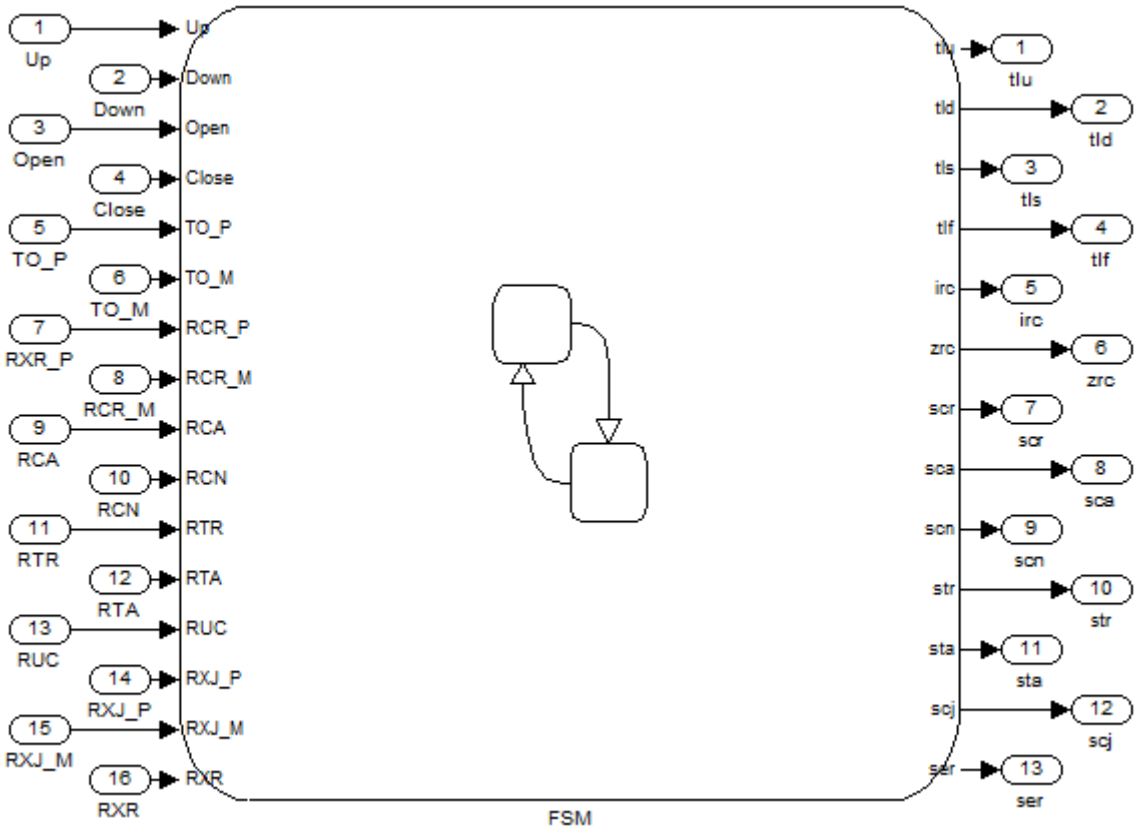


Figura 3.1: Modelo de alto nível no Simulink.

O autômato foi implementado na ferramenta Stateflow de maneira direta, pois o modelo Statechart provê a representação de estados, transições, condições e ações com construções muito similares às de um diagrama de estados tradicional. A Figura 3.2 mostra uma visão ampliada do estado *Closed*, onde podemos observar como as

transições e suas respectivas condições são modeladas, além das atribuições de valores às saídas que representam ações executadas.

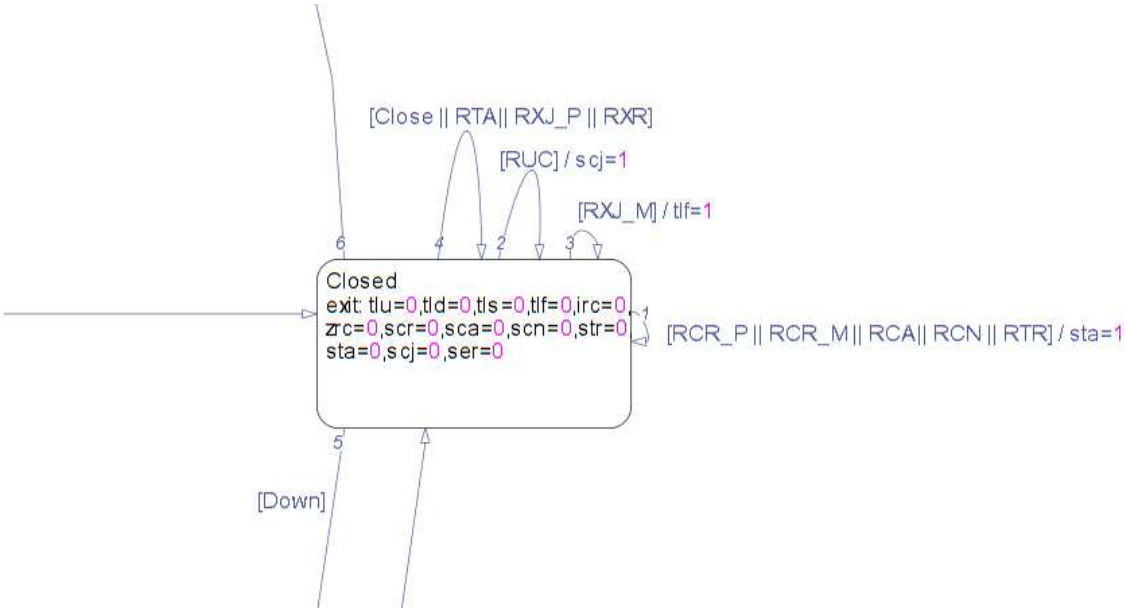


Figura 3.2: Modelagem do estado *Closed* na ferramenta Stateflow.

Ao final da implementação do autômato, a ferramenta auxiliou no processo de detecção de eventuais erros de representação no diagrama. Depois disso, com o auxílio de blocos de construção de sinais do Simulink, simulamos o cenário descrito na Seção 2.4 para testar o funcionamento do autômato. O Stateflow realça as transições ativadas na própria visualização do modelo, deixando claro para o desenvolvedor quais são as mudanças de estados, o que permite um melhor entendimento do comportamento da solução.

### 3.2.3 Geração de código

Para dar início à geração de código-fonte, configuramos uma série de opções no ambiente Simulink que interferem nesse processo, sendo duas delas as mais importantes: (i) definição do sistema alvo, *i.e.*, o conjunto de regras a ser seguido para geração do código-fonte; e (ii) a linguagem do código de saída e o seu padrão.

Primeiramente, era preciso determinar qual o alvo mais adequado, pois é ele que determina as estruturas externas que serão utilizadas e o estilo do código gerado. Dentre as diversas opções, o alvo ERT (*Real-Time Workshop Embedded Coder*) foi o selecionado, porque gera um código totalmente independente de bibliotecas de terceiros, tornando-o portável a outras plataformas. Na escolha da linguagem de programação, tivemos o intuito de preservar a simplicidade e manter um método de comparação com o código manual que obtivemos (discutido na seção 3.4), por isso selecionamos o C com o padrão ANSI C90.

Configuradas todas as opções necessárias, geramos o código C a partir do modelo do autômato. A Figura 3.3 mostra um trecho de código correspondente ao tratamento de eventos de entrada quando o autômato se encontra no estado *Starting*. O tratamento de eventos é feito por meio de estruturas condicionais e as ações são assinaladas em variáveis. É possível observar, também, marcadores em forma de comentários, ressaltados em negrito na imagem, que permitem a navegação direta das estruturas do

modelo para trechos específicos do código que as representam. Essa funcionalidade é muito útil para o desenvolvedor refatorar o modelo tendo em vista o impacto no código gerado.

```
/* Function for Stateflow: '<Root>/FSM' */
static void FSM_Closed(void)
{
  /* During 'Closed': '<S1>:169' */
  if (FSM_U.RUC) {
    /* Transition: '<S1>:175' */
    FSM_exit_atomic_Closed();
    FSM_B.scj = true;

    /* Entry 'Closed': '<S1>:169' */
    FSM_DWork.is_c1_FSM = (uint8_T)FSM_IN_Closed;
    FSM_B.state = 2;
  } else if (FSM_U.RXJ_M) {
    /* Transition: '<S1>:176' */
    FSM_exit_atomic_Closed();
    FSM_B.tlf = true;

    /* Entry 'Closed': '<S1>:169' */
    FSM_DWork.is_c1_FSM = (uint8_T)FSM_IN_Closed;
    FSM_B.state = 2;
  } else if (FSM_U.RCR_P || FSM_U.RCR_M || FSM_U.RCA || FSM_U.RCN || FSM_U.RTR)
  {
    /* Transition: '<S1>:174' */
    FSM_exit_atomic_Closed();
    FSM_B.sta = true;

    /* Entry 'Closed': '<S1>:169' */
    FSM_DWork.is_c1_FSM = (uint8_T)FSM_IN_Closed;
    FSM_B.state = 2;
  } else if (FSM_U.Close || FSM_U.RTA || FSM_U.RXJ_P || FSM_U.RXR) {
    /* Transition: '<S1>:172' */
    FSM_exit_atomic_Closed();

    /* Entry 'Closed': '<S1>:169' */
    FSM_DWork.is_c1_FSM = (uint8_T)FSM_IN_Closed;
    FSM_B.state = 2;
  } else if (FSM_U.Down) {
    /* Transition: '<S1>:187' */
    FSM_exit_atomic_Closed();

    /* Entry 'Initial': '<S1>:73' */
    FSM_DWork.is_c1_FSM = (uint8_T)FSM_IN_Initial;
    FSM_B.state = 0;
  } else {
    if (FSM_U.Open) {
      /* Transition: '<S1>:192' */
      FSM_exit_atomic_Closed();
      FSM_B.irc = true;
      FSM_B.scr = true;

      /* Entry 'Req_Sent': '<S1>:76' */
      FSM_DWork.is_c1_FSM = (uint8_T)FSM_IN_Req_Sent;
      FSM_B.state = 6;
    }
  }
}
```

Figura 3.3: Trecho do código C gerado no Simulink para o estado *Closed*.

## 3.3 Rational Rhapsody

### 3.3.1 Descrição da ferramenta

O software IBM Rational Rhapsody Developer (IBM, 2011-b) é um ambiente de modelagem baseado na Unified Modeling Language (OBJECT MANAGEMENT GROUP, 2011) para auxiliar engenheiros e desenvolvedores na criação de sistemas embarcados ou de tempo-real. Ele tem funcionalidades de validação comportamental do sistema embarcado por meio de prototipação rápida, depuração visual e execução do modelo. Para a modelagem do sistema, o Rational se vale dos diagramas de caso de uso, de estrutura estática, de colaboração e comportamentais, sendo possível gerar código a partir de vários deles.

Uma funcionalidade particularmente interessante no Rhapsody é denominada *roundtrip*. Ela permite que as modificações realizadas pelo desenvolvedor diretamente

código sejam refletidas em estruturas do modelo, mantendo sincronizadas ambas as representações da solução. Dessa forma, o usuário pode realizar pequenas ou extensas modificações onde considerar ser mais conveniente, modelo ou código, o que torna o processo como um todo mais ágil, diminuindo o tempo de desenvolvimento.

### 3.3.2 Modelagem

A versão do Rhapsody utilizada foi a *Developer for C*, adaptada para desenvolvedores que tem como objetivo a geração de código na linguagem C. O diagrama UML que utilizamos para a representação do autômato de estados finitos é chamado de UML Statechart, uma variação orientada a objetos do modelo de Harel (HAREL, 1987), cujas características já foram apresentas na Subseção 3.2.2. A Figura 3.4 representa o modelo de alto nível do controlador do protocolo LCP. A classe FSM foi criada para ter um UML Statechart associado, o qual representa o autômato de estados finitos. Além disso, ela também contém atributos simbolizando ações de saída e operações públicas representando uma interface de comunicação. Por fim, o arquivo *Events* é usado pela classe FSM para declarar as variáveis utilizadas como eventos de entrada do autômato.

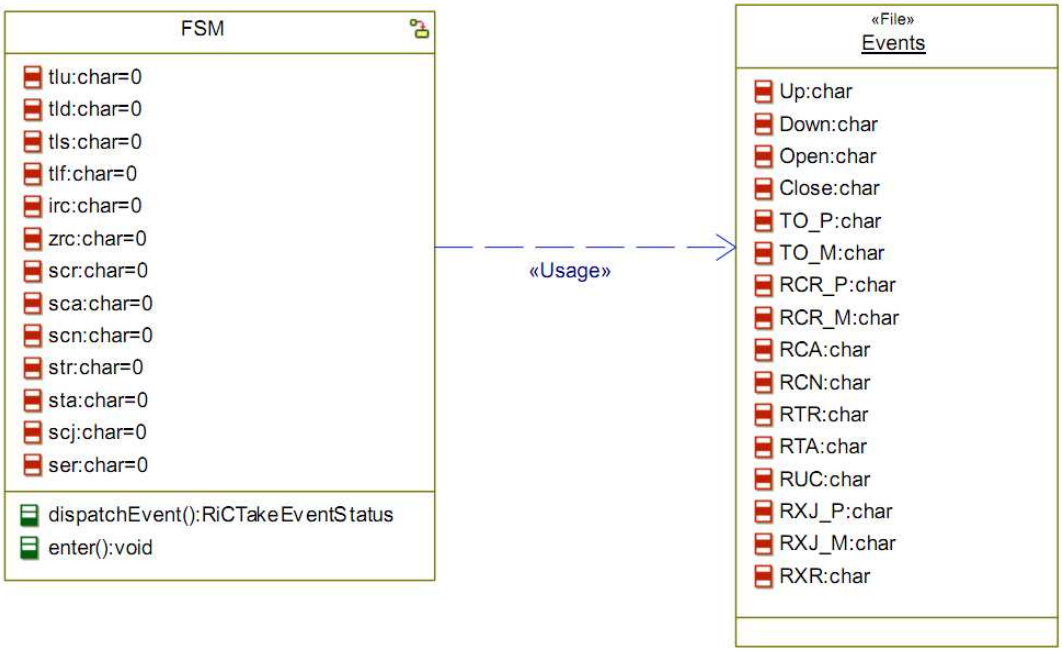


Figura 3.4: Modelo de alto nível do controlador do protocolo LCP no Rhapsody.

Assim como no Simulink, o processo de implementação do autômato no Rhapsody foi direto, pois a ferramenta disponibiliza um ambiente muito intuitivo para a criação de estados, transições, condições e ações. Contudo, uma diferença importante é a necessidade da utilização de trechos de código diretamente no modelo, como para a elaboração das condições de transição e suas ações. Assim como para o Simulink, mostramos na Figura 3.5 o modelo do estado *Closed* na ferramenta Rhapsody, contendo transições e suas condições, além das ações executadas. Podemos visualizar o trecho de código “me->tlf=1” na transição do canto inferior direito do bloco representando um estado.

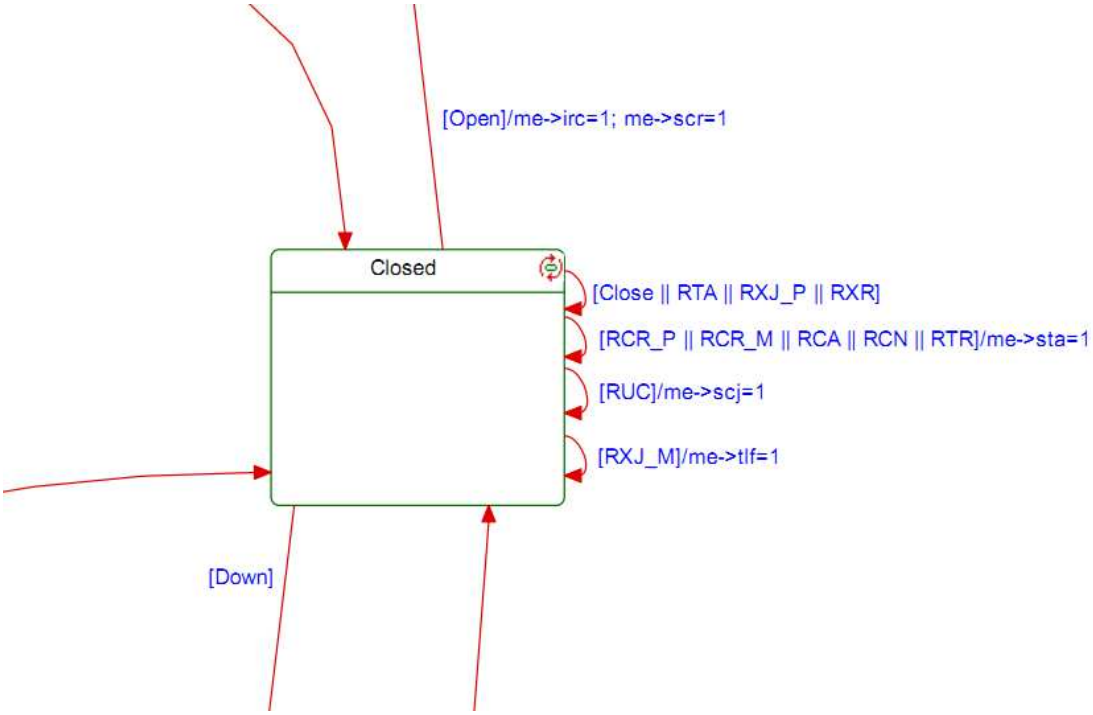


Figura 3.5: Modelagem do estado *Closed* na ferramenta Rational Rhapsody.

Concluída a modelagem do autômato, testamos a semântica das transições do diagrama com o intuito de descobrir possíveis erros na sua representação. Contudo, não foi possível realizar simulações animadas do modelo devido à falta de algumas bibliotecas, como será explicado na Subseção 3.3.3.

### 3.3.3 Geração de código

A etapa de configuração das opções de geração de código no Rhapsody foi mais complexa do que no Simulink devido ao grande número de propriedades que interferem no processo. Apesar de tomar mais tempo do desenvolvedor, essa tarefa precisa ser executada somente uma vez em um projeto grande, além de permitir que as configurações sejam reusadas em outras aplicações. Todavia, mesmo com esse refinamento controlado pelo desenvolvedor, não é possível desvincular o código gerado de um framework específico, como será explicado a seguir.

Juntamente ao modelo da aplicação e às propriedades mencionadas no parágrafo anterior, o Rhapsody também recebe como entrada, na fase de geração de código, um conjunto de bibliotecas portável e orientado a objetos chamado Object Execution Framework (IBM, 2011-a). O OXF possui abstrações de um sistema de tempo-real que estruturam o código gerado e dão significado concreto aos conceitos do UML, o que permite a tradução de modelos como o Statechart para uma linguagem de programação. Além disso, o framework também fornece a semântica básica para a simulação animada dos modelos comportamentais.

Entretanto, a licença acadêmica que obtivemos provê os arquivos do framework somente para C++ e Java. Não obstante, é possível gerar um código C contendo a semântica geral do diagrama Statechart, com a inconveniência de referências a arquivos e chamadas de funções indisponíveis. Devido à simplicidade geral do nosso sistema e analisando o código C gerado, conseguimos identificar quais eram as incompatibilidades e, então, decidimos criar os devidos arquivos. Esses arquivos

contêm variáveis, macros e assinaturas de funções, que não possuem qualquer semântica, pois o nosso projeto não depende dessas estruturas para a obtenção dos resultados. Dessa forma, conseguimos utilizar o código sem qualquer modificação, em vez de desenvolver a aplicação em C++, o que impossibilitaria uma comparação justa de determinadas métricas. Assim, foi possível compilar e executar o projeto, mas não foi possível simulá-lo na ferramenta, já que essa funcionalidade depende do framework OXF como um todo.

As Figuras 3.6 e 3.7 exibem o trecho de código correspondente ao tratamento de dos eventos de entrada do estado *Closed*, o mesmo estado detalhado anteriormente para o Simulink na Figura 3.3. Podemos identificar o uso de estruturas condicionais para verificar a ocorrência de eventos e a atribuição de valores a variáveis simbolizando as ações executadas. Do mesmo modo que na versão do Simulink, existem marcações para estabelecer uma correlação entre as partes do modelo e suas respectivas representações no código. Essas marcações, representadas em negrito nas imagens, também são utilizadas para a sincronização entre modelo e código (*roundtrip*). Por fim, observamos a utilização de uma variável do tipo “RiCTakeEventStatus” e das funções “RiCReactive_pushNullConfig” e “RiCReactive_popNullConfig”, os quais pertencem à estrutura OXF e não interferem na execução do algoritmo, como explicado no parágrafo anterior.

```
RiCTakeEventStatus FSM_ClosedTakeNull(FSM* const me) {
  RiCTakeEventStatus res = eventNotConsumed;
  /*## transition 23 */
  if(Close || RTA || RXJ_P || RXR) {
    RiCReactive_popNullConfig(&(me->ric_reactive));
    {
        /*#[ state ROOT.Closed.(Exit) */
        me->tlu=0; me->tld=0; me->tls=0; me->tlf=0; me->irc=0; me->zrc=0; me->scr=0;
        me->sca=0; me->scn=0; me->str=0; me->sta=0; me->scj=0; me->ser=0;
        /*#]*/
    }
    RiCReactive_pushNullConfig(&(me->ric_reactive));
    me->rootState_subState = FSM_Closed;
    me->rootState_active = FSM_Closed;
    {
        /*#[ state ROOT.Closed.(Entry) */
        me->state=2;
        /*#]*/
    }
    res = eventConsumed;
  } else {
    /*## transition 24 */
    if(RCR_P || RCR_M || RCA || RCN || RTR) {
      RiCReactive_popNullConfig(&(me->ric_reactive));
      {
          /*#[ state ROOT.Closed.(Exit) */
          me->tlu=0; me->tld=0; me->tls=0; me->tlf=0; me->irc=0; me->zrc=0; me->scr=0;
          me->sca=0; me->scn=0; me->str=0; me->sta=0; me->scj=0; me->ser=0;
          /*#]*/
      }
      {
          /*#[ transition 24 */
          me->sta=1;
          /*#]*/
      }
      RiCReactive_pushNullConfig(&(me->ric_reactive));
      me->rootState_subState = FSM_Closed;
      me->rootState_active = FSM_Closed;
      {
          /*#[ state ROOT.Closed.(Entry) */
          me->state=2;
          /*#]*/
      }
      res = eventConsumed;
    } else {
        /*## transition 25 */
      if(RUC) {
        RiCReactive_popNullConfig(&(me->ric_reactive));
        {
            /*#[ state ROOT.Closed.(Exit) */
            me->tlu=0; me->tld=0; me->tls=0; me->tlf=0; me->irc=0; me->zrc=0; me->scr=0;
            me->sca=0;  me->scn=0; me->str=0; me->sta=0; me->scj=0; me->ser=0;
            /*#]*/
        }
      }
```

Figura 3.6: Trecho do código C gerado no Rhapsody para o estado *Closed*. (Parte 1).

```
        {
            /*#[ transition 25 */
            me->scj=1;
            /*#]*/
        }
        RiCReactive_pushNullConfig(&(me->ric_reactive));
        me->rootState_subState = FSM_Closed;
        me->rootState_active = FSM_Closed;
        {
            /*#[ state ROOT.Closed.(Entry) */
            me->state=2;
            /*#]*/
        }
        res = eventConsumed;
      } else {
        /*## transition 26 */
        if(RXJ_M) {
          RiCReactive_popNullConfig(&(me->ric_reactive));
          {
              /*#[ state ROOT.Closed.(Exit) */
              me->tlu=0; me->tld=0; me->tls=0; me->tlf=0; me->irc=0; me->zrc=0;
              me->scr=0; me->sca=0; me->scn=0; me->str=0; me->sta=0; me->scj=0;
              me->ser=0;
              /*#]*/
          }
          {
              /*#[ transition 26 */
              me->tlf=1;
              /*#]*/
          }
          RiCReactive_pushNullConfig(&(me->ric_reactive));
          me->rootState_subState = FSM_Closed;
          me->rootState_active = FSM_Closed;
          {
              /*#[ state ROOT.Closed.(Entry) */
              me->state=2;
              /*#]*/
          }
          res = eventConsumed;
        } else {
          /*## transition 69 */
          if(Down) {
            RiCReactive_popNullConfig(&(me->ric_reactive));
            {
                /*#[ state ROOT.Closed.(Exit) */
                me->tlu=0; me->tld=0; me->tls=0; me->tlf=0; me->irc=0; me->zrc=0;
                me->scr=0; me->sca=0; me->scn=0; me->str=0; me->sta=0; me->scj=0;
                me->ser=0;
                /*#]*/
            }
            RiCReactive_pushNullConfig(&(me->ric_reactive));
            me->rootState_subState = FSM_Initial;
            me->rootState_active = FSM_Initial;
            {
                /*#[ state ROOT.Initial.(Entry) */
                me->state=0;
                /*#]*/
            }
            res = eventConsumed;
          } else {
            /*## transition 83 */
            if(Open) {
              RiCReactive_popNullConfig(&(me->ric_reactive));
              {
                  /*#[ state ROOT.Closed.(Exit) */
                  me->tlu=0; me->tld=0; me->tls=0; me->tlf=0; me->irc=0; me->zrc=0;
                  me->scr=0; me->sca=0;  me->scn=0; me->str=0; me->sta=0; me->scj=0;
                  me->ser=0;
                  /*#]*/
              }
              {
                  /*#[ transition 83 */
                  me->irc=1; me->scr=1;
                  /*#]*/
              }
              RiCReactive_pushNullConfig(&(me->ric_reactive));
              me->rootState_subState = FSM_Req_Sent;
              me->rootState_active = FSM_Req_Sent;
              {
                  /*#[ state ROOT.Req_Sent.(Entry) */
                  me->state=6;
                  /*#]*/
              }
              res = eventConsumed;
            }
          }
        }
      }
    }
  }
  return res;
}
```

Figura 3.7: Trecho do código C gerado no Rhapsody para o estado *Closed*. (Parte 2).

## 3.4 Código escrito manualmente

Para estabelecer um método de comparação com o desenvolvimento automatizado da solução para o controlador do protocolo LCP, fez-se necessário um código escrito manualmente para referência. Obtivemos um código em linguagem C para o pacote de controle do protocolo PPP do sistema Dragonfly BSD (DRAGONFLY, 2011). Neste pacote, identificamos e estudamos os arquivos responsáveis pelo processamento do autômato de estados finitos do LCP.

Este código tem uma estrutura diferente daqueles gerados pelas ferramentas Simulink e Rhapsody. Ao invés de uma função para cada estado que testa o evento de entrada, existe uma função para cada evento de entrada que testa o estado atual. Essa diferença se deve à representação do modelo nas ferramentas para geração automática, as quais agrupam estados ao invés de eventos.

Depois de estudarmos o código detalhadamente, decidimos remover os trechos que não representam qualquer funcionalidade para o nosso estudo. Assim, o código passou a conter somente o processamento de cada evento e a geração das ações de saída, o que permite uma comparação mais acurada entre as diferentes soluções. Como o código possui um agrupamento por eventos em vez de estados, decidimos não incluir uma imagem nos moldes das Figuras 3.3, 3.6 e 3.7, pois ela não resultaria em uma comparação fidedigna em relação aos códigos-fonte do Simulink ou do Rhapsody. Nesse caso, optamos por inserir todo o código-fonte modificado no Apêndice A.

# 4 AVALIAÇÃO

## 4.1 Introdução

Neste capítulo, descreveremos os critérios de comparação das soluções empregados neste trabalho. Apresentaremos o ambiente de simulação utilizado para a execução das diferentes versões do controlador do protocolo LCP. Além disso, descreveremos o experimento conduzido. Por fim, detalharemos e analisaremos os resultados obtidos, realizando comparações da qualidade de cada uma das soluções para o domínio dos sistemas embarcados.

## 4.2 Critérios de Comparação

O estudo comparativo a que nos propomos neste trabalho exige que sejam definidos critérios de avaliação das soluções que sejam representativos no domínio em análise. Neste sentido, materializamos esses critérios em métricas particularmente importantes para os sistemas embarcados. Os itens a seguir revisitam essas métricas justificando sua importância nesse contexto e explicam como elas serão computadas nesta avaliação:

- **Memória total:** como a disponibilidade de memória é limitada em sistemas embarcados, ela se torna um fator de restrição de projeto relevante. A memória total utilizada pela aplicação será a soma da memória de programa e da memória de dados.
- **Energia consumida:** a energia consumida pela execução da aplicação é muito importante devido ao fato dos sistemas embarcados terem fontes de alimentação com capacidade de armazenamento limitada. O cômputo da energia consumida por cada solução pode ser obtido pelo somatório do consumo médio de energia por instrução na arquitetura alvo multiplicado pelo número de vezes em que a instrução foi executada pela aplicação.
- **Número de instruções:** o número de instruções executadas por uma aplicação em embarcados está fortemente vinculado à presença de requisitos de sistemas de tempo-real. Esses requisitos definem restrições quanto ao tempo de atendimento a solicitações (*i.e.,* o tempo máximo para a finalização de uma determinada operação). Como métrica do tempo total necessário para a conclusão de uma tarefa, utilizaremos o número de instruções executadas pelo processo para a sua realização.
- **Abstração:** na engenharia de software, a abstração busca definir a solução de um problema com representações próximas da sua semântica, escondendo detalhes de implementação, o que facilita o desenvolvimento e a manutenção do sistema. Assim, o programador pode otimizar o seu trabalho mantendo o foco em um

número menor de conceitos a cada instante. A discussão deste critério não levará em conta uma métrica específica, mas realizará uma comparação entre os modelos de representação do controlador do protocolo LCP nas diferentes soluções.

- **Potencial de reúso:** o reúso de código tem como objetivo economizar tempo de desenvolvimento por meio da redução de trabalho redundante. Assim, o potencial de reúso está relacionado à capacidade de adequação de um código para uma nova solução. Vamos estabelecer uma comparação desse potencial avaliando o nível de acoplamento da solução gerada a bibliotecas externas.
- **Tempo de desenvolvimento:** como discutido na contextualização deste trabalho, um dos fatores que impulsiona o domínio dos sistemas embarcados é o tempo de chegada do produto ao mercado. Portanto o tempo de desenvolvimento de uma solução é muito importante nesse contexto.

## 4.3 Ambiente de Simulação

Com o intuito de obter informações disponíveis somente durante a execução de um programa, *i.e.*, os itens (i), (ii) e (iii) da seção 4.2, utilizamos um ambiente de simulação chamado SIMICS. O SIMICS (MAGNUSSON, P. et al, 2002) é uma ferramenta que permite a simulação de sistemas computacionais reais por meio da emulação de uma arquitetura física. A descrição dessa arquitetura pode variar desde um sistema simples com unidade central de processamento e memória, até um sistema complexo contendo diversas placas e unidades funcionais. Durante a emulação de uma arquitetura, é possível criar um histórico dos eventos de hardware ocorridos, por exemplo, execuções de instruções, acessos à memória e acessos a dispositivos de entrada e saída.

Um trabalho desenvolvido por membros do Laboratório de Sistemas Embarcados do Instituto de Informática da UFRGS utilizou um modelo fornecido pelo fabricante para mapear no SIMICS um sistema computacional completo para o processador MIPS R4000 (MIRAPURI, 1992), o qual roda o núcleo do sistema operacional Linux na versão 2.4.2. Além disso, o grupo do laboratório também desenvolveu uma aplicação (RUTZIG, 2009) que toma como entrada o histórico de eventos de hardware fornecido pelo SIMICS e uma especificação do consumo médio de energia por instrução do processador MIPS e gera um relatório do total de instruções executadas e do consumo de energia. Dessa forma, é possível executar o código de uma determinada aplicação em um ambiente controlado e computar as métricas físicas mencionadas anteriormente, as quais dificilmente poderiam ser aferidas no sistema real correspondente.

## 4.4 Descrição do Experimento

Com o objetivo de obter as métricas físicas da execução do controlador do protocolo LCP nas três diferentes versões da solução, criamos um experimento que toma como entrada uma sequência de eventos e gera uma saída contendo todas as transições executadas. Neste experimento, cerca de mil eventos são enviados ao controlador para que cada transição do autômato de estados finitos seja acionada pelo menos uma vez. Criamos este cenário hipotético para testar todas as transições do autômato, pois, segundo dados obtidos de históricos de eventos de sistemas reais, na maioria dos casos, tanto a negociação de opções como o encerramento da conexão do enlace acontecem graciosamente. Assim, apenas uma pequena parte das transições do autômato seria

coberta em um cenário real, o que não era desejado. A sequência de eventos e as respectivas transições de saída deste cenário estão no Apêndice B.

Para processar os eventos de entrada, as três diferentes soluções foram compiladas tendo como alvo o conjunto de instruções do processador MIPS R4000. Empregamos o método de ligação dinâmica, pois observamos que a ligação estática sobrecarregava o código objeto com diversas bibliotecas utilizadas somente para dar suporte à execução nessa arquitetura, aumentando consideravelmente o segmento de código do arquivo executável, acabando por impossibilitar comparações fiéis do tamanho da memória de instruções. Além disso, durante a compilação, desativamos todas as possíveis otimizações para minimizar a influência do funcionamento do compilador nos resultados obtidos. Os códigos objeto gerados foram executados pelo simulador SIMICS e os dados relativos ao número de instruções e energia consumida foram relatados pela aplicação mencionada na Seção 4.3.

Além desses dados, utilizamos informações dos arquivos contendo o código executável das diferentes soluções para obter os consumos de memória. Cinco diferentes valores foram somados para obtermos o valor total da memória alocada: (i) o tamanho do segmento de código; (ii) o tamanho do segmento de dados; (iii) o tamanho do segmento de variáveis globais não inicializadas; (iv) o tamanho máximo da memória dinâmica do programa; e (v) o tamanho máximo atingido pela pilha do programa. Os valores dos itens (i), (ii) e (iii) podem ser extraídos facilmente de cada arquivo executável. Visto que somente uma solução utiliza memória dinâmica (*i.e.*, item (iv)), identificamos os pontos de seu código onde esses recursos eram alocados e os computamos no total da memória de dados. Já para o cálculo do item (v), para cada solução, analisamos o código de cada função e, devido ao baixo número de chamadas encadeadas e a inexistência de chamadas recursivas, obtivemos o tamanho ocupado pelas estruturas em memória e computamos o valor máximo atingido pela pilha do programa.

A Tabela 4.1 sumariza as métricas físicas obtidas para cada uma das soluções. O valor no campo Memória de Programa leva em consideração somente o item (i), enquanto que o campo Memória de Dados soma os valores dos itens (ii), (iii), (iv) e (v). As colunas de variação apresentam o sobrecusto percentual da solução à esquerda em relação à versão manual.

Tabela 4.1: Métricas físicas das três diferentes soluções.

| *Métrica* | *Manual* | *Simulink* | *Variação* | *Rhapsody* | *Variação* |
|---|---|---|---|---|---|
| Memória de Programa (Bytes) | 10.404 | 18.154 | **75%** | 40.249 | **287%** |
| Memória de Dados (Bytes) | 390 | 432 | **11%** | 555 | **42%** |
| Energia (μJ) | 96.594 | 100.017 | **3,6%** | 99.723 | **3,2%** |
| Número de Instruções | 189.400.325 | 196.112.609 | **3,6%** | 195.534.777 | **3,2%** |

## 4.5 Análise dos Resultados

Analisando a Tabela 4.1, podemos observar que o fator predominante no consumo de memória é o tamanho do código de máquina. Isso se deve ao modelo de computação ser orientado a controle (*control-flow*), ou seja, a aplicação é um sistema reativo que responde a eventos do ambiente em que está inserida gerando um conjunto de valores de saída. Além disso, vemos que a solução do Rhapsody consumiu aproximadamente 286% mais de memória do que a solução escrita manualmente. Dependendo das restrições do sistema embarcado, essa grande diferença pode impossibilitar a adoção da solução automatizada. Por outro lado, a solução do Simulink tem um sobrecusto em relação à escrita manual de 74%, uma penalidade mais admissível se considerarmos os ganhos detalhados nos parágrafos seguintes.

Ainda observando a Tabela 4.1, percebemos que a melhor solução para o consumo de energia também é aquela referente ao código escrito manualmente. Entretanto, vemos que a diferença entre o consumo das três soluções não ultrapassa 3,6%. Este mesmo índice é obtido quando comparamos os dados da última linha da Tabela 4.1, o que evidencia a forte correlação entre o número de instruções e a energia consumida.

Este resultado era esperado devido à característica dos algoritmos do modelo de computação orientado a controle e à arquitetura superescalar do processador MIPS R4000. Todas as três soluções, as quais implementam esse modelo, contém diversas estruturas condicionais e algumas atribuições de variáveis. Esses trechos de código das diferentes soluções tendem a se materializar em um mesmo número total de instruções da linguagem de máquina. Por outro lado, como em um processador superescalar todas as unidades funcionais são acionadas para qualquer instrução executada, o consumo de energia de cada instrução é praticamente o mesmo. Assim, as soluções tendem a exibir um comportamento de consumo de energia proporcional ao número de instruções executadas de acordo com uma mesma constante. Por fim, a diferença reduzida entre o melhor e o pior caso para essas duas métricas demonstra a eficiência energética e o bom desempenho em número de instruções das soluções de geração automática para este estudo de caso.

A abstração provida pelo desenvolvimento da solução em um modelo de alto nível teve um grande impacto na implementação e na simulação do sistema. Primeiramente, a instanciação direta do modelo contido na especificação do protocolo em um diagrama nas ferramentas de automação foi significativamente mais fácil do que a modificação do código escrito manualmente, pois os conceitos de eventos, ações e transição de estados ficaram muito mais claros. Além disso, já no final da modelagem realizamos a busca automática por possíveis inconsistências semânticas, simplificando um processo que se torna exaustivo na escrita manual. Por fim, a simulação com entidades visuais na ferramenta Simulink proporcionou um claro entendimento do funcionamento do protocolo, diferentemente da análise de entradas e saídas necessária na versão manual.

Comparando as soluções obtidas das ferramentas de automação, observamos que o potencial de reúso do Simulink é maior do que aquele do Rhapsody. Conforme visto na Seção 3.3.3, o código gerado pelo Rhapsody é fortemente acoplado à plataforma OXF. Tendo isso em vista, foi necessário um esforço considerável para fazer com que o código-fonte gerado fosse compilável em um ambiente desprovido das bibliotecas OXF. Portanto, apesar do nosso sucesso na obtenção de uma solução executável, não é possível garantir que isto seria reprodutível para outros casos, indicando que o reúso dessa solução dependerá da portabilidade de bibliotecas externas à arquitetura alvo. Por

outro lado, a solução obtida do Simulink é independente de funcionalidades externas, o que permite que esta mesma implementação possa ser mapeada para outras plataformas.

As métricas do domínio físico ou de qualidade de software mencionadas anteriormente passam a adquirir maior representatividade quando existem ganhos no tempo de desenvolvimento, pois esse fator está diretamente relacionado ao tempo de chegada do produto ao mercado. O tempo de modelagem do autômato nas ferramentas de automação, medido por nós durante a execução dessa tarefa, foi praticamente o mesmo, já que não há diferenças na construção desse formalismo. Infelizmente não temos dados disponíveis sobre o tempo de escrita manual do código original. Entretanto, somente a tarefa de torná-lo equivalente aos códigos gerados automaticamente consumiu cerca de 80% do tempo da modelagem completa do autômato em cada ferramenta. Neste sentido, desconsiderando o tempo para adequação (*i.e.*, curva de aprendizagem) do projetista às ferramentas de automação utilizadas neste trabalho, o qual tende a se diluir com a criação de novos projetos, estimamos que o tempo total de escrita do código seria consideravelmente maior do que a modelagem direta do autômato.

# 5 CONCLUSÃO

A demanda por novas funcionalidades em sistemas embarcados têm crescido nos últimos anos. Para atender a essas expectativas, os projetistas tem dado preferência ao desenvolvimento de software, em detrimento do desenvolvimento de hardware, devido, principalmente, à flexibilidade, ao menor custo, à facilidade de atualização do sistema e à capacidade de reúso. Por outro lado, a complexidade do desenvolvimento desse software embarcado tem crescido, demandando o emprego de processos e metodologias que lidem com ela e que sejam eficientes na exploração das características do hardware. Outro fator complicador é a existência de janelas de chegada dos produtos embarcados ao mercado (*time-to-market*) cada vez menores, pressionando por um aumento da produtividade.

No cenário exposto acima, a síntese automática de software aparece como uma alternativa. Neste processo, são construídas visões particulares das aplicações por meio da utilização de modelos formais ou semi-formais. O código é, então, gerado diretamente desses modelos, pois eles possuem semânticas próprias. Os ganhos principais com a utilização da síntese automática estão relacionados ao aumento da produtividade proporcionado pela maior abstração, além das facilidades para verificação e simulação das aplicações, o que permite a detecção e a correção de problemas ainda em estágios iniciais do projeto.

No presente trabalho, aplicamos a síntese automática de software a partir do modelo formal de um protocolo de comunicação. Utilizando ferramentas comerciais, modelamos o autômato de estados finitos que descreve o funcionamento do protocolo, simulamos um cenário de estabelecimento de conexão (apenas e uma ferramenta) e geramos código-fonte. Depois disso, essas soluções e aquela escrita manualmente foram compiladas para a realização de um experimento, o qual exercitou todas as transições do autômato com o objetivo de obtermos dados sobre as métricas físicas das soluções.

Analisando os resultados obtidos com o experimento e tomando a versão de escrita manual como referência, observamos que a solução da ferramenta Rhapsody possui uma penalidade muito significativa no consumo de memória, ao contrário do Simulink, que apresenta um sobrecusto muito menor, mas ainda representativo. Por outro lado, o número de instruções executadas e o consumo de energia de todas as soluções estão muito próximos.

Os altos níveis de abstração providos pelas soluções de síntese automática também representaram ganhos significativos durante a modelagem, pois facilitaram a detecção das falhas, além de disponibilizarem mecanismos para a simulação das soluções. Ao final de todo o processo, pudemos avaliar o tempo consumido para a modelagem nas ferramentas de síntese automática e compará-lo com aquele gasto para a modificação do código escrito manualmente. Essa avaliação aponta para um cenário em que a geração

automática de código-fonte leva menos tempo para ser executada, o que se mostra como uma das principais vantagens da utilização dessa técnica.

Por fim, cabe ressaltar que a estrutura de sistema orientado a controle destacada para o LCP está presente também em qualquer protocolo de comunicação. Logo, podemos inferir que os resultados obtidos com esse estudo de caso podem ser generalizados, ou seja, acreditamos que a análise feita anteriormente também seja valida para outros protocolos, mas a prova dessas afirmações fica a título de trabalhos futuros.

# REFERÊNCIAS

ABDULLAH, I.; MENASCE, D. Protocol Specification and Automatic Implementation Using XML and CBSE. **Proceedings of CIIT"03**, nov. 2003.

DRAGONFLY. **Dragonfly BSD**. Disponível em: <http://www.dragonflybsd.org/>. Acesso em: jun. 2011.

EBERT, C.; SALECKER. J. Embedded Software – Technologies and Trends. **IEEE Software**, v.26, n.3, p. 14-18, jun. 2009.

GOMAA, H. **Designing Concurrent Distributed, and Real-Time Applications with UML**. Reading, MA: Addison-Wesley, 2000.

GUNAWAN, E. Selection of a formal description technique (FDT) for a FDT based protocol converter. **Singapore ICCS/ISITA '92**. **'Communications on the Move'**, [S.l.], v.1, p. 188-192, Nov 1992. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=254977>. Acesso em: jun. 2011.

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of Computer Programming**. v.8, n.3, p. 231-274, jun. 1987. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/0167642387900359>. Acesso em: jun. 2011.

IBM. **Object Execution Framework**. Disponível em: <http://publib.boulder.ibm.com/infocenter/rhaphlp/v7r6/index.jsp?topic=/com.ibm.rhp.frameworks.doc/topics/rhp_t_fw_working_object_execution_framewk.html>. Acesso em: jun 2011.

IBM. **Rational Rhapsody**. Disponível em: <http://www-01.ibm.com/software/awdtools/rhapsody/>. Acesso em: jun. 2011.

MAGNUSSON, P.; CHRISTENSSON, M.; ESKILSON, J.; FORSGREN, D.; HÅLLBERG, G.; HÖGBERG, J.; LARSSON, F.; MOESTEDT, A.; WERNER, B. SIMICS: A Full System Simulation Platform. **IEEE Computer**, v.35, n.2, p. 50-58, fev. 2002.

MIRAPURI, S.; WOODACRE, M.; VASSEGHI, N.; The Mips R4000 processor. **Micro, IEEE**, v.12, n.2, p. 10-22, abr. 1992.

OBJECT MANAGEMENT GROUP. **UML – Unified Modeling Language**. Disponível em <http://www.uml.org>. Acesso em: jun. 2011.

RUTZIG, M.; BECK FILHO, A.; CARRO, L. Dynamically Adapted Low Power ASIPs. **International Workshop on Reconfigurable Computing**, Karlsruhe, 2009.

SELIC, B. Models, Software Models and UML. In: LAVAGNO, L.; MARTIN, G.; SELIC, B. (Ed.). **UML for Real**: Design of Embedded Real-Time Systems. Boston: Kluwer Academic, 2003. p. 1-16.

SIMPSON, W. **The Point-to-Point Protocol (PPP)**: RFC 1661. [S.l.]: Internet Engineering Task Force, Network Working Group, 1994.

SOMMERVILLE, I. **Software Engineering**. 8th ed., Reading: Addison-Wesley, 2006.

THE MATHWORKS. **Real-Time Workshop**. Disponível em: <http://www.mathworks.com/products/simulink-coder/>. Acesso em: jun. 2011.

THE MATHWORKS. **Simulink v. 7.7**. Disponível em: <http://www.mathworks.com/products/simulink/>. Acesso em: jun. 2011.

THE MATHWORKS. **Stateflow R2011a**. Disponível em: <http://www.mathworks.com/products/stateflow/>. Acesso em: jun. 2011.

TURLEY, J.: **The Two Percent Solution**. Dez. 2002. Disponível em: < http://www.eetimes.com/discussion/other/4024488/The-Two-Percent-Solution>. Acesso em: jun. 2011.

VENTURE DEVELOPMENT CORPORATION. 2010 Embedded System Engineering Survey. **Market Intelligence Service**, Natick, MA, USA. set. 2010.

ZIMMERMANN, H. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. **Communications, IEEE Transactions on**, v.28, n.4, p. 425-432, abr. 1980.

# GLOSSÁRIO

| | |
|---|---|
| Datagrama | Unidade de transmissão da camada de rede. Um datagrama pode ser encapsulado em um ou mais pacotes passados ao nível de enlace. |
| Quadro | Unidade de transmissão da camada de enlace. |
| Pacote | Unidade básica de encapsulamento, a qual é passada pela interface entre a camada de rede e a camada de enlace. |

# ANEXO <ARTIGO TG1: ESTUDO COMPARATIVO ENTRE A SÍNTESE DE SOFTWARE MANUAL E AUTOMÁTICA PARA PROTOCOLOS DE COMUNICAÇÃO EM SISTEMAS EMBARCADOS >

# Estudo Comparativo entre a Síntese de Software Manual e Automática para Protocolos de Comunicação em Sistemas Embarcados

**Fabrício Girardi Andreis, Luigi Carro**

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{fgandreis,carro}@inf.ufrgs.br

***Abstract.*** *Embedded systems contain the majority of processors sold worldwide and demand more and more software, which are growing in complexity. In order to tackle this issue, methodologies that provide higher levels of abstraction and techniques for the process of automated synthesis are needed. Moreover, the proliferation of subsystems, which demand coordination through communication protocols, may also be observed. The implementation of these protocols for embedded systems must meet constraints and requirements of this domain. Therefore, in this work we carry out a study of the existing tools for protocol automated synthesis and we propose metrics to compare the code generated manually and automatically.*

***Resumo.*** *Os sistemas embarcados contém a grande maioria dos processadores vendidos no mundo e demandam cada vez mais software, os quais estão crescendo em complexidade. Para lidar com esse problema, são fundamentais metodologias que forneçam maior abstração e técnicas para a automatização do processo de síntese. Pode ser observada também a proliferação de subsitemas, os quais exigem coordenação por meio de protocolos de comunicação. A implementação desses protocolos para os sistemas embarcados deve atender às restrições e requisitos desse domínio. Portanto, nesse trabalho são estudadas ferramentas de síntese automática de protocolos existentes e são propostas métricas para a comparação entre o código gerado manual e automaticamente.*

## Introdução

Os computadores pessoais e servidores representam uma fatia extremamente reduzida do total de processadores comercializados no mundo atualmente, representando menos de 2% do total [Turley 2002]. Os demais processadores estão presentes nas soluções de sistemas embarcados, os quais, assim como computadores convencionais, demandam o desenvolvimento de software. Além disso, devido aos avanços em tecnologias de informação e comunicação, a tendência é de que, no futuro, os domínios de aplicação desses sistemas sejam ampliados, já que cada vez mais produtos devem passar a executar algum tipo de software [Ebert 2009].

Do ponto de vista de projeto, o software para sistemas embarcados apresenta restrições e requisitos funcionais e não-funcionais. Existem significativas limitações impostas pelo hardware, como o tamanho da memória disponível, o consumo de energia e a capacidade de processamento. Alguns sistemas para aplicações classificadas como críticas para a vida, principalmente dos setores de transporte e saúde, requerem também

a adequação a padrões elevados de confiabilidade e segurança. Além disso, o ciclo de vida dos produtos embarcados tem se tornado cada vez mais reduzido, fazendo com que o tempo de projeto deva ser encurtado para atender a janelas de chegada ao mercado (*time-to-market*) cada vez menores, atingindo a média de apenas 16 meses [VDC 2010].

Os sistemas embarcados começaram como soluções para problemas pequenos e específicos, em que a camada de software disponibilizava uma funcionalidade específica e atuava diretamente no acionamento e na configuração do hardware. Entretanto, nos últimos anos, a demanda por novas funcionalidades nos produtos embarcados vem crescendo rapidamente. Essa evolução pode ser claramente percebida em mercados como o de telefones celulares, em que cada nova geração de aparelhos apresenta inovações significativas. Nesse contexto, o software representa a parte principal do projeto de sistemas embarcados. Essa preferência pelo software pode ser explicada por fatores como o menor custo em relação a uma solução complexa em hardware, a sua flexibilidade e possibilidade de adaptabilidade a mudanças nos requisitos ainda na fase de projeto, o reuso de blocos de projetos anteriores e a facilidade de atualização do sistema. Por outro lado, esse processo tem aumentado a complexidade do próprio software, demandando o emprego de processos e metodologias que lidem com ela, mas eficientes ao ponto de explorarem da melhor maneira possível as características do hardware.

Estatísticas recentes mostram que mais de 40% dos projetos de sistemas embarcados encontram-se atrasados [VDC 2010], o que está relacionado com o aumento de complexidade e a diminuição da janela de chegada ao mercado. Os esforços dos projetistas para lidar com a complexidade do projeto concentram-se no uso de técnicas e ferramentas baseadas em especificações com maior abstração. Essas medidas estão em consonância com as visões defendidas por autores como [Selic 2003] e [Gomma 2000], de que essa abordagem é o único meio viável para lidar com a complexidade das novas gerações de sistemas embarcados. Para a adequação às janelas de chegada ao mercado reduzidas, uma das abordagens possíveis é a utilização de ferramentas de síntese automática para a geração de código a partir de modelos de maior abstração.

O processo de síntese consiste, basicamente, na geração de um artefato a partir de sua especificação, a qual foi possivelmente descrita com um nível mais alto de abstração. Na síntese automática de software, o código é gerado automaticamente, não sendo mais escrito pelo desenvolvedor. Uma das implicações da síntese é que o processo de modelagem deve ser menos custoso do que a escrita manual do código. O objetivo principal dessa técnica é o aumento da produtividade dos desenvolvedores por meio do deslocamento da atuação desses profissionais para níveis mais altos de abstração. Como a síntese é realizada de maneira automática, ela pode se beneficiar de mais eficientemente das características do hardware alvo, no caso de sistemas embarcados, ao se aproveitar de técnicas de exploração de espaço de projeto [Gries 2003].

A tendência da presença de software embarcado em diversos sistemas [Ebert 2009] vem acompanhada pela proliferação de subsistemas, cada um com diferentes processadores. Essa especialização é comumente associada à demanda por paralelização da execução das tarefas. Além disso, a distribuição da execução por diversos processadores projetados para tarefas específicas, os quais são mais eficientes

energeticamente, é a tendência para substituição das arquiteturas com um único processador executando em frequências altas [Goodwin 2007].

Para que haja coordenação entre esses diversos processadores, são necessários meios de comunicação entre eles, o que pode ser modelado por protocolos de comunicação. Como supracitado, com a utilização de diversos processadores específicos para tarefas determinadas, o desenvolvimento de software torna-se ainda mais complexo, o que prejudica o *time-to-market*, apesar do ganho em desempenho e energia. Nesse sentido, faz-se essencial o uso de síntese automática de software, especialmente para os protocolos de comunicação, os quais são críticos para o funcionamento do sistema como um todo.

Levando em consideração todos os aspectos discutidos nos parágrafos anteriores, o objetivo deste trabalho é a realização de um estudo comparativo entre o código gerado manualmente e aquele gerado pelas diferentes ferramentas para síntese automática de software para sistemas embarcados. A aplicação a ser implementada será um protocolo de comunicação especificado em um formalismo amplamente aceito, que são as máquinas de estado finitas (descritas na seção 3.1). Para isso, são propostos critérios de comparação entre o código escrito pelo desenvolvedor e aquele sintetizado pela ferramenta.

## 2. Revisão bibliográfica

Esta seção discute alguns conceitos e trabalhos relacionados à automação de software e à síntese automática de protocols que servirão de base para o desenvolvimento do restante do trabalho. A seção 2.1 faz uma breve introdução de algumas metodologias e paradigmas de engenharia de software que oferecem suporte à automação. Na seção 2.2 são apresentados trabalhos relacionados à automação de software para sistemas embarcados. Por fim, a seção 2.3 discute trabalhos relacionados à síntese automática de protocolos.

### 2.1 O suporte da engenharia de software para a automação

Na área de engenharia de software, as ferramentas CASE (*Computer-aided software engineering*) são amplamente utilizadas para aumentar a abstração e automatizar o desenvolvimento de software, incluindo mecanismos para a geração de código a partir de modelos. No período inicial da adoção dessa tecnologia, o código gerado possuia grande complexidade devido ao mapeamento ineficiente entre a descrição de alto nível e a plataforma alvo da aplicação. O cenário atual avançou consideravelmente, principalmente devido ao uso de linguagens orientadas a objetos e à utilização de plataformas de reuso. Entretanto, o crescimento da complexidade dessas plataformas demanda muito esforço para sua adequação ao projeto em desenvolvimento, o que volta a sobrecarregar o desenvolvedor com a escrita e a manutenção de código.

Um das abordagens correntes para tratar das limitações de automatização descritas acima é a Engenharia Orientada a Modelos (MDE). Essa metodologia tem seu enfoque na criação de modelos (ou abstrações) mais próximos dos conceitos de algum domínio particular em vez de focar o desenvolvimento em conceitos específicos do ambiente de execução. A MDE é estruturada sobre dois elementos básicos: as linguagens específicas de domínio (DSLs) e as máquinas de transformação. DSLs definem o relacionamento entre conceitos e descrevem semânticas e restrições

específicos de um domínio, provendo alta abstração somente dentro desse contexto, o que possibilita a utilização de premissas fundamentais nas etapas de geração de código. As máquinas de transformação tratam da conversão de modelos para outros artefatos, por exemplo, código-fonte, entradas para simulação ou até mesmo outros modelos [Schimidt 2006].

A metodologia utilizada pela MDE aproveita o poder de representação dos conceitos de um domínio específico, provido pelas DSLs, para sucessivamente transformar modelos em outros modelos ou em código-fonte. Além da facilidade obtida pela utilização de elementos gráficos diretamente relacionados ao domínio alvo, outra vantagem importante é a possibilidade de detecção de erros em estágios iniciais do projeto por meio da verificação de modelos e da imposição de restrições nesse contexto específico.

A abordagem proposta pelo OMG (Object Management Group) para utilização da metodologia MDE é a Arquitetura Dirigia por Modelos (MDA) [OMG 2003]. Em MDA, o conceito de mapeamento entre modelos estabelece mecanismos automáticos ou semi-automáticos para a transformação de modelos independentes de plataforma (PIM) em modelos dependentes de plataforma (PSM). Os modelos PIM e PSM são expressos em linguagens de modelagem cujas sintaxes abstrata e concreta e semântica operacional são definidas por meio de *meta-modelos*. A geração de código a partir de modelos de alto nível pode ser vista como a transformação sucessiva de PIMs em um PSM, o qual é usado para a derivação da implementação real, ou seja, a transformação de *meta-modelos* em um código-fonte de programa em uma linguagem específica.

## 2.2 Síntese de software para sistemas embarcados

Uma abordagem para a execução de modelos e síntese de código baseada em MDA foi descrita em [Schattkowsky 2005]. Nesse trabalho, é apresentada uma plataforma para execução de modelos (MEP, do inglês *Model Execution Platform*), baseada em um subconjunto da UML 2 (Unified Modeling Language) com semânticas comportamentais bem definidas, que provê conceitos abstratos para a geração de código para sistemas embarcados.

O trabalho desenvolvido em [Wehrmeister 2008] trata do problema da geração automática de código para Sistemas Embarcados Distribuídos de Tempo Real (DERTS), a qual tem que considerar requisitos funcionais e não-funcionais, utilizando conceitos da Arquitetura Dirigida por Modelos e do paradigma de Orientação a Aspectos. É proposta uma ferramenta para geração de código chamada GenERTiCA. Essa ferramenta possui como entrada um modelo UML e o transforma em um modelo independente de plataforma (PIM), o qual contem informações sobre a estrutura, o comportamento e o tratamento de requisitos não-funcionais dos DERTS. A ferramenta procura gerar o código mais completo possível (i.e., não somente o esqueleto de classes) para várias linguagens alvo por meio de scripts, os quais materializam as regras de transformação entre modelos. Em [Moreira 2010], os autores apresentam uma expansão da ferramenta GenERTiCA para a transformação de UML em VHDL, uma linguagem de descrição de hardware.

ForSyDe [Sander 2004] é um *framework* para a síntese de sistemas embarcados baseada na linguagem Haskell. Partindo de um modelo de especificação formal, o qual captura a funcionalidade do sistema em um alto nível de abstração, a ferramenta realiza

um processo de refinamento até chegar em um modelo de implementação que é otimizado para a síntese. Para prover abstração para diversos domínios, ForSyDe requer o conhecimento e classificação de vários modelos de computação diferentes, o que pode aumentar o tempo de desenvolvimento.

### 2.3 Síntese de procotolos de comunicação

Em [Liu 1999], protocolos de comunicação são modelados por meio de máquinas de estados finitos (FSMs, descritas na seção 3.1) para a geração de esqueletos de código C++. O código representa somente os diferentes estados e a troca de mensagens entre as entidades, ficando a cargo do desenvolvedor a sua implementação. Já em [Abdullah 2003], é proposta uma linguagem baseada em XML para a especificação e a geração automática de código para protocolos de comunicação.

Em [Balarin 1999], é apresentada uma metodologia para a síntese de software para sistemas embarcados de tempo real, baseados em um formalismo chamado de Máquinas de Estados Finitos Concorrentes, proposto pelos autores. A especificação formal contida nessa proposta permite que algoritmos de verificação formal baseados em FSMs sejam utilizados. O código é gerado na linguagem C.

O trabalho desenvolvido em [Siegmund 2002] apresenta uma metodologia de projeto para a síntese de hardware de controle de protocolos de comunicação baseda em descrições declarativas. Os autores propõem um formalismo que é um subconjunto da linguagem SystemC, permitindo a especificação, verificação e síntese de protocolos no contexto da descrição do sistema. A metodologia foi testada por meio da síntese dos controladores de alguns protocolos em descrições de hardware em nível RTL (*Register Transfer Language*) .

## 3. Síntese de software a partir de Máquinas de Estado Finitas (FSM)

### 3.1 Especificação de protocolos

Um protocolo de comunicação consiste basicamente de entidades orientadas a eventos que se comunicam por meio da troca de mensagens. O comportamento de uma entidade pode ser descrito em termos de transições de estados que ela executa em resposta a eventos internos e externos. O modelo mais amplamente aceito para a representação de um protocolo de comunicação é a máquina de estados finitos (FSM) [Gunawan 1992]. Existem também variações de FSM como a CFSM (*Communicating Finite-State Machine*) definidas especificamente para protocolos de comunicação.

### 3.2 Critérios para comparação entre ferramentas de geração automática

Para que possamos melhor avaliar as ferramentas para geração automática de software a partir de máquinas de estado finitas (FSM), alguns critérios comparativos são estabelecidos e descritos nos tópicos seguintes:

- **Suporte direto à representação de máquina de estados:** uma máquina de estados finitos é um diagrama comportamental. Nesse critério as ferramentas são classificadas quanto ao suporte direto à especificação de uma máquina de estados ou se deve ser usada uma representação intermediária, o que requer a transformação da FSM em um equivalente.

- **Linguagem do código de saída:** a linguagem do código gerado representa um critério importante quanto à aplicabilidade da solução, pois ela deve ser compatível com o fluxo de desenvolvimento do projeto alvo.
- **Suporte à simulação do modelo:** a simulação do modelo FSM pode auxiliar o projetista a detectar falhas na especificação do protocolo, facilitando o processo de verificação e validação.
- **Suporte à otimização do código de saída:** o código gerado automaticamente tende a ser menos eficiente do que aquele escrito manualmente. Por isso, é importante verificar se as ferramentas disponibilizam formas de otimização do código gerado, já que essa etapa é fundamental para se atingir a performance esperada.
- **Suporte à verificação automática do código gerado:** esse critério diz respeito à existência de funcionalidades de verificação integradas à ferramenta, ou seja, avalia a possibilidade do desenvolvedor verificar se o código gerado atende às especificações materializadas no modelo do projeto. Esse aspecto se mostra importante para diminuir o custo de uma eventual verificação manual da solução gerada.

### 3.3 Ferramentas de geração automática de software a partir de FSM

Ferramentas de síntese automatica de software vão desde simples implementações que traduzem máquinas de estado finitas (FSM) para código C, C# ou Java [Smart State Studio 2010], até complexas soluções com funcionalidades de simulação, verificação e otimização do código. Nessa subseção serão apresentadas três soluções comerciais para a geração de código a partir de especificações de máquinas de estado finitas.

Simulink [The Mathworks 2010, Simulink] é um ambiente gráfico para projeto baseado em modelos e simulação de sistemas embarcados e dinâmicos. Ele provê um ambiente interativo e um conjunto personalizável de bibliotecas de blocos que permitem o projeto, simulação, implementação e teste de diversos sistemas, como os de comunicação, processamento de sinais, processamento de vídeo e processamento de imagens. Existem também pacotes extras para geração de código, teste, verificação e validação. O Real-Time Workshop [The Mathworks 2010, Real Time Workshop] é um gerador de código C ou C++ a partir de modelos Simulink e Stateflow. A Stateflow permite a especificação de máquinas de estados diretamente em seus modelos internos, suporta simulação do modelo, tem funcionalidades para a otimização do código de saída e verificação do código gerado.

IBM Rational Rhapsody [IBM 2010] é um ambiente de desenvolvimento visual para auxiliar engenheiros de sistemas e desenvolvedores de software na criação de sistemas embarcados e de tempo real e no projeto e desenvolvimento de software. A ferramenta utiliza conceitos de MDA, suportando modelos difundidos na indústra como UML e SysML. A versão voltada para engenheiros de sistema possibilita a simulação de modelos para a validação comportamental e a descrição de máquinas de estado é feita pelo modelo específico disponível no UML 2. Há suporte à geração de código C, C++, Java ou Ada e à execução de tarefas de verificação. Também é dado suporte à otimização do código gerado.

Scade [Esterel Technologies 2010] é mais um ambiente gráfico para modelagem, simulação e síntese de software embarcado. Ele é adequado para aplicações

críticas por gerar código qualificável para padrões de segurança e confiabilidade comuns no domínio de sistemas aviação e automotivo. São disponibilizadas bibliotecas de blocos da mesma forma que o Stateflow. A ferramenta também suporta simulação e verificação baseada em modelos. A síntese de software pode ser feita diretamente de especificações de FSM e o código C gerado é otimizado e independente de plataforma.

**Tabela 1. Resumo da classificação das ferramentas apresentadas**

| | **Stateflow** | **IBM Rational Rhapsody** | **Scade** |
|---|---|---|---|
| Suporte direto à representação de máquina de estados | Sim | Sim | Sim |
| Linguagem do código de saída | C, C++ | C, C++, Java e Ada | C |
| Suporte à simulação do modelo | Sim | Sim | Sim |
| Suporte à otimização do código de saída | Sim | Sim | Sim |
| Suporte à verificação do código gerado | Sim | Sim | Sim |

A Tabela 1 mostra que as três ferramentas atendem completamente aos critérios estabelecidos na subseção 3.2. Dessa forma, consideramos que o desenvolvimento do restante do trabalho é factível com a utilização das ferramentas apresentadas.

### 3.4 Critérios para comparação entre o código gerado manual e automaticamente

Para que possamos realizar a comparação entre o código escrito manualmente e aquele sintetizado automaticamente, precisamos avaliar os resultados gerados segundo critérios representativos para o domínio em análise. Neste trabalho, nós materializamos os critérios em forma de métricas com alta relevância para os sistemas embarcados [Oliveira 2008], as quais são apresentadas a seguir:

**i)** **Energia consumida:** esse critério tem grande relevância em sistemas embarcados, pois eles costumam ser alimentados por baterias, as quais tem capacidade de armazenamento de energia limitada.

**ii)** **Memória total:** a memória total ocupada pelo programa gerado pode ser crucial na escolha de uma solução para sistemas os quais a disponibilidade de memória é um fator de restrição de projeto.

**iii)** **Desempenho:** o desempenho está fortemente ligado ao atendimento de requisistos de sistemas de tempo real, muito comuns em embarcados.

**iv)** **Abstração:** a abstração é uma das maneiras encontradas pela engenharia de software para lidar com a complexidade do software, o que facilita o desenvolvimento e a manutenção, sendo, portanto, importante também para sistemas embarcados.

**v)** **Complexidade do código:** com essa métrica, vamos tentar medir a adequação do código para manutenção e depuração. Essa métrica está totalmente

relacionada à facilidade do desenvolvedor em entender as estruturas e o funcionamento do programa.

vi) **Tempo de desenvolvimento:** como mencionado anteriormente, um dos fatores que impulsiona o domínio dos sistemas embarcados é o tempo de chegada do produto ao mercado [Graaf 2003]. Portanto o tempo de desenvolvimento de uma solução é muito importante nesse contexto. Por estar relacionada com as horas trabalhadas dos desenvolvedores, essa métrica pode representar, também, parte do custo do projeto.

vii) **Potencial de reuso:** o código escrito ou gerado para ser reusado tende a ser mais genérico, *i.e.*, menos otimizado para o escopo de um projeto, o que o tornaria proibitivamente ineficiente para algumas aplicações de sistemas embarcados. A métrica tem por objetivo capturar o potencial de reutilização da solução levando em conta essa problemática.

Pode-se perceber que as três primeiras métricas são relativas às imposições do hardware dos sistemas embarcados, enquanto que as demais estão relacionadas com o processo de desenvolvimento do projeto como um todo.

## 4. Análise

Levando em consideração a revisão da literatura realizada, as definições de critérios comparativos e a pesquisa por ferramentas capazes de gerar código automaticamente, pode-se observar que existem, tanto na academia quanto na indústria, soluções passíveis de serem usadas na próxima etapa do trabalho, a fim de se executar a comparação proposta.

Todas as ferramentas para automação de software detalhadas na seção 3.3 já foram obtidas com licenças acadêmicas, as quais são suficientes para a realização dos estudos de caso. O tempo para a ambientação com cada ambiente de desenvolvimento pode representar um fator complicador, mas acreditamos que há tempo suficiente para extrairmos dados que embasem a comparação proposta, já que, como ponto positivo, todos permitem a representação direta de máquinas de estados finitos.

A definição da especificação dos diferentes protocolos a serem implementados será realizada logo no início do desenvolvimento da segunda etapa do trabalho. Isso será feito de maneira a alcançarmos um equilíbrio entre os objetivos finais do estudo e o tempo para executá-lo.

## 5. Conclusão

A maioria dos processadores atuais está presente em sistemas embarcados e a tendência é que a demanda por software para esses sistemas continue crescendo, em um mercado impulsionado por fatores como custo, confiabilidade e tempo de projeto. A produção de software para esse domínio já é considerada um gargalo significativo. Um aspecto complicador é que as práticas tradicionais de engenharia de software não podem ser aplicadas diretamente a esse contexto, pois elas não consideram os requisitos não-funcionais, essenciais para a especificação dos embarcados. Entretanto, novas abordagens e ferramentas tem surgido para auxiliar os projetistas a combater esses problemas.

Nesse sentido, a automação da síntese de software se faz necessária tanto para adequar os projetos a janelas de chegada ao mercado cada vez menores como para atender requisitos não-funcionais por vezes conflitantes com boas práticas de programação [Wagner 2009]. Além disso, existe uma tendência de ramificação de unidades funcionais especializadas pelo sistema. Para que haja coordenação, é preciso prover meios para a realização de troca de informações entre elas, o que requer a implementação de protocolos de comunicação em nível de software.

No contexto exposto acima, o presente trabalho faz um levantamento bibliográfico sobre as áreas de automação de software para sistemas embarcados e de síntese automática de protocolos de comunicação. Foi proposto um estudo comparativo entre o código escrito manualmente e aquele gerado automaticamente, que deverá analisar métricas específicas das soluções obtidas com essas duas técnicas. As ferramentas para a geração do código automático estão descritas na seção 3.3.

A segunda etapa do presente trabalho, a ser executada durante o primeiro semestre de 2011, consiste na implementação manual e automática de um protocolo de comunicação para sistemas embarcados e no estudo comparativo entre o código gerado por essas técnicas. A tabela abaixo define um cronograma de atividades para o cumprimento dos objetivos propostos e discutidos neste texto.

**Tabela 2. Cronograma de atividades para o restante do trabalho no ano de 2011.**

| | **Março** | **Abril** | **Maio** | **Junho** |
|---|---|---|---|---|
| Ambientação com as ferramentas para geração automática de código a partir de FSMs | X | | | |
| Modelagem do(s) protocolo(s) nos formalismos das ferramentas | X | X | | |
| Geração automatizada de código a partir das ferramentas | | X | X | |
| Escrita manual do código | | X | X | |
| Comparação entre os códigos gerados manual e automaticamente | | | X | X |
| Redação do trabalho final | | | X | X |

## Referências

Abdullah, I. e Menascé, D. (2003) “Protocol specification and automatic implementation using XML and CBSE”. IASTED conference on Communications, Internet and Information Technology.

Balarin, F. *et al.* (1999) “Synthesis of Software Programs for Embedded Control Applications”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 18 (6), 834-849.

Ebert, C. e Salecker, J. (2009) “Embedded Software – Technologies and Trends”. IEEE Software, 26 (3), 14-18.

Esterel Technologies. (2010) Scade v. 6. Disponível em http://www.esterel-technologies.com/products/scade-suite, último acesso em novembro de 2010.

Graaf, B. Lormans, M. and Toetenel, H. (2003) “Embedded software engineering: The state of practice”. IEEE Software, 20 (6), 61-69.

Gomaa, H. (2000) “Designing Concurrent Distributed, and Real-Time Applications with UML.” Reading, MA: Addison-Wesley.

Goodwin, D., Rowen, C. e Martin, G. 2(2007) “Configurable Multi-Processor Platforms for Next Generation Embedded Systems”. Design Automation Conference. ASP-DAC '07. Asia and South Pacific, pp. 744-746.

Gries, M. (2003) “Methods for Evaluating and Covering the Design Space During Early Design Development”. Integration, the VLSI Journal. [S.l.], v. 38, pp. 131-183.

IBM. (2010) IBM Rational Rhapsody . Disponível em http://www.ibm.com/developerworks/rational/products/rhapsody, último acesso em novembro de 2010.

Liu, C.S. (1999) “A Program Generator for Object-Based Implementation of Communication Protocol Software”, Proceedings of the The Fourth International Symposium on Autonomous Decentralized Systems – ISADS.

Oliveira, M. F. S *et al*. (2008) “Software Quality Metrics and their Impact on Embedded Software”. 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software - MOMPES, vol. 5, pp. 68-77.

OMG. (2003) MDA Guide Version 1.0.1. Disponível em http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf, último acesso em novembro de 2010.

OMG. (2010) UML – Unified Modeling Language. Disponível em http://www.uml.org, último acesso em novembro de 2010.

Sander, I. e Jantsch, A. (2004) “System Modeling and Transformational Design Refinement in ForSyDe”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23 (1), 17-32.

Schmidt, D.C. (2006) “Model-driven Engineering”. IEEE Computer, 39 (2), 25-31.

Selic, B. (2003) “Models, Software Models and UML”, UML for Real: Design of Embedded Real-Time Systems, L. Lavagno, G. Martin e B. Selic. Boston: Kluwer Academic, p. 1-16.

Siegmund, R. e Müller, D. (2002) “A novel synthesis technique for communication controller hardware from declarative data communication protocol specifications”. 39th Conf. on Design Automation.

Smart State Studio. (2010) Disponível em http://www.smartstatestudio.com, último acesso em novembro de 2010.

The Mathworks. (2010) Real-Time Workshop. Disponível em: http://www.mathworks.com/products/rtw, último acesso em novembro de 2010.

The Mathworks. (2010) Simulink. Disponível em: http://www.mathworks.com/products/simulink, último acesso em novembro de 2010.

Turley, J. (2002) “The Two Percent Solution”. Disponível em http://www.eetimes.com/discussion/other/4024488/The-Two-Percent-Solution, último acesso em novembro de 2010.

Venture Development Corporation. (2010). “2010 Embedded System Engineering Survey”. Market Intelligence Service, Natick, MA, USA.

Wagner, F. e Carro, L. (2009) “Metodologias e Técnicas de Engenharia de Software para Sistemas Embarcados”. JAI’09 – XXIII Jornadas de Atualização de Informática, Capítulo 4. SBC/PUC, Bento Gonçalves - RS.

Wehrmeister, M.A., Freitas, E.P., Pereira, C.E. e Ramming, F.J. (2008) “GenERTiCA: A Tool for Code Generation and Aspects Weaving”. In: 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing – ISORC (pp. 234-238). Los Alamitos: IEEE Computer Society.

# APÊNDICE A <CÓDIGO DO CONTROLADOR DO PROTOCOLO LCP ESCRITO MANUALMENTE>

```
/*
 *  fsm.h  -  {Link,  IP}  Control  Protocol  Finite  State  Machine
definitions.
 *
 * Copyright (c) 1989 Carnegie Mellon University.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted
 * provided that the above copyright notice and this paragraph are
 * duplicated in all such forms and that any documentation,
 * advertising materials, and other materials related to such
 * distribution and use acknowledge that the software was developed
 * by Carnegie Mellon University.  The name of the
 * University may not be used to endorse or promote products derived
 * from this software without specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
 * WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 * $FreeBSD: src/usr.sbin/pppd/fsm.h,v 1.7 1999/08/28 01:19:03 peter
Exp $
 *  $DragonFly:  src/usr.sbin/pppd/fsm.h,v  1.3  2003/11/03  19:31:40
eirikn Exp $
 */

/*
 * Each FSM is described by an fsm structure and fsm callbacks.
 */
typedef struct fsm_T {
      char tlu;
      char tld;
      char tls;
      char tlf;
      char irc;
      char zrc;
      char scr;
      char sca;
      char scn;
      char str;
      char sta;
      char scj;
      char ser;
      char passive;
      char restart;
      char cross;
```

```
    char state;
} fsm_T;


/*
 * Link states.
 */
#define INITIAL         0     /* Down, hasn't been opened */
#define STARTING  1     /* Down, been opened */
#define CLOSED          2     /* Up, hasn't been opened */
#define STOPPED         3     /* Open, waiting for down event */
#define CLOSING         4     /* Terminating the connection, not open
*/
#define STOPPING  5     /* Terminating, but open */
#define REQSENT         6     /* We've sent a Config Request */
#define ACKRCVD         7     /* We've received a Config Ack */
#define ACKSENT         8     /* We've sent a Config Ack */
#define OPENED          9     /* Connection available */

#define __TO_M__  0
#define __TO_P__  1
#define __RCR_M__ 0
#define __RCR_P__ 1
#define __RXJ_M__ 0
#define __RXJ_P__ 1

/*
 * Prototypes
 */
void fsm_init(fsm_T *);
void fsm_lowerup(fsm_T *);
void fsm_lowerdown(fsm_T *);
void fsm_open(fsm_T *);
void fsm_close(fsm_T *);
void fsm_timeout(fsm_T *, char);
void fsm_rconfreq(fsm_T *, char);
void fsm_rconfack(fsm_T *);
void fsm_rconfnakrej(fsm_T *);
void fsm_rtermreq(fsm_T *);
void fsm_rtermack(fsm_T *);
void fsm_runkcode(fsm_T *);
void fsm_rcoderej(fsm_T *, char);
void fsm_recho(fsm_T *f);

/*
 * fsm.c - {Link, IP} Control Protocol Finite State Machine.
 *
 * Copyright (c) 1989 Carnegie Mellon University.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted
 * provided that the above copyright notice and this paragraph are
 * duplicated in all such forms and that any documentation,
 * advertising materials, and other materials related to such
 * distribution and use acknowledge that the software was developed
 * by Carnegie Mellon University.  The name of the
 * University may not be used to endorse or promote products derived
 * from this software without specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
```

```
 * WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 *  $FreeBSD:  src/usr.sbin/pppd/fsm.c,v  1.8  1999/08/28  01:19:02  peter
Exp $
 *   $DragonFly:   src/usr.sbin/pppd/fsm.c,v   1.3   2003/11/03   19:31:40
eirikn Exp $
 */
#include "fsm.h"
void clear_outputs(fsm_T *f) {
      f->tlu = 0;
      f->tld = 0;
      f->tls = 0;
      f->tlf = 0;
      f->irc = 0;
      f->zrc = 0;
      f->scr = 0;
      f->sca = 0;
      f->scn = 0;
      f->str = 0;
      f->sta = 0;
      f->scj = 0;
      f->ser = 0;
      f->passive = 0;
      f->restart = 0;
      f->cross = 0;
}

/*
 * fsm_init - Initialize fsm.
 *
 * Initialize fsm state.
 */
void
fsm_init(fsm_T *f)
{
    f->state = INITIAL;
      clear_outputs(f);
}


/*
 * fsm_lowerup - The lower layer is up.
 */
void
fsm_lowerup(fsm_T *f)
{
      clear_outputs(f);
    switch( f->state ){
    case INITIAL:
      f->state = CLOSED;
      break;

    case STARTING:
      /* Send an initial configure-request */
      f->irc = 1;
      f->scr = 1;
      f->state = REQSENT;
      break;
    }
}
```

```
/*
 * fsm_lowerdown - The lower layer is down.
 *
 * Cancel all timeouts and inform upper layers.
 */
void
fsm_lowerdown(fsm_T *f)
{
      clear_outputs(f);
    switch( f->state ){
    case CLOSED:
      f->state = INITIAL;
      break;

    case STOPPED:
      f->tls = 1;
      f->state = STARTING;
      break;

    case CLOSING:
      f->state = INITIAL;
      break;

    case STOPPING:
    case REQSENT:
    case ACKRCVD:
    case ACKSENT:
      f->state = STARTING;
      break;

    case OPENED:
      f->tld = 1;
      f->state = STARTING;
      break;
    }
}


/*
 * fsm_open - Link is allowed to come up.
 */
void
fsm_open(fsm_T *f)
{
      clear_outputs(f);
    switch( f->state ){
    case INITIAL:
      f->tls = 1;
      f->state = STARTING;
      break;

    case CLOSED:
      f->irc = 1;
      f->scr = 1;
      f->state = REQSENT;
      break;

    case CLOSING:
```

```
      f->state = STOPPING;
      /* fall through */
    case STOPPED:
      case STOPPING:
    case OPENED:
      f->restart = 1;
      break;
    }
}


/*
 * fsm_close - Start closing connection.
 *
 * Cancel timeouts and either initiate close or possibly go directly
to
 * the CLOSED state.
 */
void
fsm_close(fsm_T *f)
{
      clear_outputs(f);
    switch( f->state ){
    case STARTING:
      f->tlf = 1;
      f->state = INITIAL;
      break;
    case STOPPED:
      f->state = CLOSED;
      break;
    case STOPPING:
      f->state = CLOSING;
      break;

    case OPENED:
      f->tld = 1;
      /* fall through */
      case REQSENT:
    case ACKRCVD:
    case ACKSENT:
      f->irc = 1;
      f->str = 1;
      f->state = CLOSING;
      break;
    }
}


/*
 * fsm_timeout - Timeout expired.
 */
void
fsm_timeout(fsm_T *f, char type)
{
      clear_outputs(f);
      if (type == __TO_P__) {
            switch (f->state) {
            case CLOSING:
            case STOPPING:
            f->str = 1;
```

```
            break;

            case REQSENT:
            case ACKRCVD:
            f->scr = 1;
            f->state = REQSENT;
            break;

            case ACKSENT:
            f->scr = 1;
            break;
            }
      } else if (type == __TO_M__) {
            switch (f->state) {
            case CLOSING:
            f->tlf = 1;
            f->state = CLOSED;

            case STOPPING:
            f->tlf = 1;
            f->state = STOPPED;
            break;

            case REQSENT:
            case ACKRCVD:
            case ACKSENT:
            f->tlf = 1;
            f->passive = 1;
            f->state = STOPPED;
            break;
            }
      }
}

/*
 * fsm_rconfreq - Receive Configure-Request.
 */
void
fsm_rconfreq(fsm_T *f, char type)
{
    clear_outputs(f);
      if (type == __RCR_P__) {
            switch( f->state ){
            case CLOSED:
            f->sta = 1;
            break;

            case STOPPED:
            f->irc = 1;
            f->scr = 1;
            f->sca = 1;
            f->state = ACKSENT;
            break;

            case REQSENT:
            case ACKSENT:
            f->sca = 1;
            f->state = ACKSENT;
            break;
```

```
            case ACKRCVD:
            f->sca = 1;
            f->tlu = 1;
            f->state = OPENED;
            break;

            case OPENED:
            f->tld = 1;
            f->scr = 1;
            f->sca = 1;
            f->state = ACKSENT;
            break;
            }
      } else if (type == __RCR_M__) {
            switch( f->state ){
            case CLOSED:
            f->sta = 1;
            break;

            case STOPPED:
            f->irc = 1;
            f->scr = 1;
            f->sca = 1;
            f->state = REQSENT;
            break;

            case REQSENT:
            case ACKSENT:
            f->scn = 1;
            f->state = REQSENT;
            break;

            case ACKRCVD:
            f->scn = 1;
            break;

            case OPENED:
            f->tld = 1;
            f->scr = 1;
            f->scn = 1;
            f->state = REQSENT;
            break;
            }
      }
}


/*
 * fsm_rconfack - Receive Configure-Ack.
 */
void
fsm_rconfack(fsm_T *f)
{
    clear_outputs(f);
    switch (f->state) {
    case CLOSED:
    case STOPPED:
      f->sta = 1;
      break;
```

```
    case REQSENT:
      f->irc = 1;
      f->state = ACKRCVD;
      break;

    case ACKRCVD:
      f->scr = 1;
      f->cross = 1;
      f->state = REQSENT;
      break;

    case ACKSENT:
      f->irc = 1;
      f->tlu = 1;
      f->state = OPENED;
      break;

    case OPENED:
      f->tld = 1;
      f->scr = 1;
      f->cross = 1;
      f->state = REQSENT;
      break;
    }
}


/*
 * fsm_rconfnakrej - Receive Configure-Nak or Configure-Reject.
 */
void
fsm_rconfnakrej(fsm_T *f)
{
    clear_outputs(f);
    switch (f->state) {
    case CLOSED:
    case STOPPED:
      f->sta = 1;
      break;

    case REQSENT:
      f->irc = 1;
      f->scr = 1;
      break;

    case ACKRCVD:
      f->scr = 1;
      f->cross = 1;
      f->state = REQSENT;
      break;

      case ACKSENT:
      f->irc = 1;
      f->scr = 1;
      break;

    case OPENED:
      f->tld = 1;
      f->scr = 1;
      f->cross = 1;
```

```
      f->state = REQSENT;
      break;
    }
}


/*
 * fsm_rtermreq - Receive Terminate-Req.
 */
void
fsm_rtermreq(fsm_T *f)
{
    clear_outputs(f);
    f->sta = 1;
      switch (f->state) {
    case ACKRCVD:
    case ACKSENT:
      f->state = REQSENT;           /* Start over but keep trying */
      break;

    case OPENED:
      f->tld = 1;
      f->zrc = 1;
      f->state = STOPPING;
      break;
    }
}


/*
 * fsm_rtermack - Receive Terminate-Ack.
 */
void
fsm_rtermack(fsm_T *f)
{
      clear_outputs(f);
    switch (f->state) {
    case CLOSING:
      f->tlf = 1;
      f->state = CLOSED;
      break;

      case STOPPING:
      f->tlf = 1;
      f->state = STOPPED;
      break;

    case ACKRCVD:
      f->state = REQSENT;
      /* fall through */
    case OPENED:
      f->tld = 1;
      f->scr = 1;
      break;
    }
}

/*
 * fsm_runkcode - Receive an Unknown-Code.
 */
```

```
void
fsm_runkcode(fsm_T *f)
{
      clear_outputs(f);
      f->scj = 1;
}

/*
 * fsm_rcoderej - Receive an Code-Reject.
 */
void
fsm_rcoderej(fsm_T *f, char type)
{
      clear_outputs(f);
    if (type == __RXJ_P__) {
            if( f->state == ACKRCVD )
                  f->state = REQSENT;
      } else if (type == __RXJ_M__) {
            switch (f->state) {

            case CLOSED:
            case CLOSING:
            f->tlf = 1;
            f->state = CLOSED;
            break;

            case STOPPED:
            case STOPPING:
            case REQSENT:
            case ACKRCVD:
            case ACKSENT:
            f->tlf = 1;
            f->state = STOPPED;
            break;

            case OPENED:
            f->tld = 1;
            f->irc = 1;
            f->str = 1;
            f->state = STOPPING;
            break;
            }
      }
}

/*
 * fsm_recho - Receive an Echo-Request or an Echo-reply or an Discard-
Request.
 */
void
fsm_recho(fsm_T *f)
{
      clear_outputs(f);
      if (f->state == OPENED)
            f->ser = 1;
}
```

# APÊNDICE B <CENÁRIO DE SIMULAÇÃO >

Este apêndice contém um cenário de simulação do autômato de estados finitos do LCP composto por uma lista de eventos de entrada e as correspondentes transições executadas. Esta sequência faz com que todas as transições sejam executadas pelo menos uma vez. As linhas estão no formato “Evento_de_Entrada: Estado_Atual => Novo_Estado”.

Down: Initial => Initial
Close: Initial => Initial
Up: Initial => Closed
Down: Closed => Initial
Close: Initial => Initial
Down: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
Open: Initial => Starting
Down: Starting => Starting
Close: Starting => Initial
Close: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
TO+: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
TO-: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
RCR+: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
RCR-: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
RCA: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
RCN: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
RTR: Initial => Initial

Down: Initial => Initial
Close: Initial => Initial
RTA: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
RUC: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
RXJ+: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
RXJ-: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
RXR: Initial => Initial
Down: Initial => Initial
Close: Initial => Initial
Open: Initial => Starting
Up: Starting => ReqSent
Down: ReqSent => Starting
Close: Starting => Initial
Open: Initial => Starting
Down: Starting => Starting
Down: Starting => Starting
Close: Starting => Initial
Open: Initial => Starting
Open: Starting => Starting
Down: Starting => Starting
Close: Starting => Initial
Open: Initial => Starting
Close: Starting => Initial
Down: Initial => Initial
Close: Initial => Initial
Open: Initial => Starting
TO+: Starting => Starting
Down: Starting => Starting
Close: Starting => Initial
Open: Initial => Starting
TO-: Starting => Starting
Down: Starting => Starting
Close: Starting => Initial
Open: Initial => Starting
RCR+: Starting => Starting
Down: Starting => Starting
Close: Starting => Initial
Open: Initial => Starting
RCR-: Starting => Starting
Down: Starting => Starting
Close: Starting => Initial

```
Open:       Initial => Starting
RCA:        Starting => Starting
Down:       Starting => Starting
Close:      Starting => Initial
Open:       Initial => Starting
RCN:        Starting => Starting
Down:       Starting => Starting
Close:      Starting => Initial
Open:       Initial => Starting
RTR:        Starting => Starting
Down:       Starting => Starting
Close:      Starting => Initial
Open:       Initial => Starting
RTA:        Starting => Starting
Down:       Starting => Starting
Close:      Starting => Initial
Open:       Initial => Starting
RUC:        Starting => Starting
Down:       Starting => Starting
Close:      Starting => Initial
Open:       Initial => Starting
RXJ+:       Starting => Starting
Down:       Starting => Starting
Close:      Starting => Initial
Open:       Initial => Starting
RXJ-:       Starting => Starting
Down:       Starting => Starting
Close:      Starting => Initial
Open:       Initial => Starting
RXR:        Starting => Starting
Down:       Starting => Starting
Close:      Starting => Initial
Up:         Initial => Closed
Up:         Closed => Closed
Down:       Closed => Initial
Close:      Initial => Initial
Up:         Initial => Closed
Down:       Closed => Initial
Down:       Initial => Initial
Close:      Initial => Initial
Up:         Initial => Closed
Open:       Closed => ReqSent
Down:       ReqSent => Starting
Close:      Starting => Initial
Up:         Initial => Closed
Close:      Closed => Closed
Down:       Closed => Initial
Close:      Initial => Initial
Up:         Initial => Closed
```

TO+: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
TO-: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
RCR+: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
RCR-: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
RCA: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
RCN: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
RTR: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
RTA: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
RUC: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
RXJ+: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
RXJ-: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
RXR: Closed => Closed
Down: Closed => Initial
Close: Initial => Initial
Up: Initial => Closed
Open: Closed => ReqSent

```
RCR+:   ReqSent => AckSent
TO-:    AckSent => Stopped
Up:     Stopped => Stopped
Down:   Stopped => Starting
Close:  Starting => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
TO-:    AckSent => Stopped
Down:   Stopped => Starting
Down:   Starting => Starting
Close:  Starting => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
TO-:    AckSent => Stopped
Open:   Stopped => Stopped
Down:   Stopped => Starting
Close:  Starting => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
TO-:    AckSent => Stopped
Close:  Stopped => Closing
Down:   Closing => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
TO-:    AckSent => Stopped
TO+:    Stopped => Stopped
Down:   Stopped => Starting
Close:  Starting => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
TO-:    AckSent => Stopped
TO-:    Stopped => Stopped
Down:   Stopped => Starting
Close:  Starting => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
TO-:    AckSent => Stopped
RCR+:   Stopped => AckSent
Down:   AckSent => Starting
Close:  Starting => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
```

RCR+: ReqSent => AckSent
TO-: AckSent => Stopped
RCR-: Stopped => ReqSent
Down: ReqSent => Starting
Close: Starting => Initial
Up: Initial => Closed
Open: Closed => ReqSent
RCR+: ReqSent => AckSent
TO-: AckSent => Stopped
RCA: Stopped => Stopped
Down: Stopped => Starting
Close: Starting => Initial
Up: Initial => Closed
Open: Closed => ReqSent
RCR+: ReqSent => AckSent
TO-: AckSent => Stopped
RCN: Stopped => Stopped
Down: Stopped => Starting
Close: Starting => Initial
Up: Initial => Closed
Open: Closed => ReqSent
RCR+: ReqSent => AckSent
TO-: AckSent => Stopped
RTR: Stopped => Stopped
Down: Stopped => Starting
Close: Starting => Initial
Up: Initial => Closed
Open: Closed => ReqSent
RCR+: ReqSent => AckSent
TO-: AckSent => Stopped
RTA: Stopped => Stopped
Down: Stopped => Starting
Close: Starting => Initial
Up: Initial => Closed
Open: Closed => ReqSent
RCR+: ReqSent => AckSent
TO-: AckSent => Stopped
RUC: Stopped => Stopped
Down: Stopped => Starting
Close: Starting => Initial
Up: Initial => Closed
Open: Closed => ReqSent
RCR+: ReqSent => AckSent
TO-: AckSent => Stopped
RXJ+: Stopped => Stopped
Down: Stopped => Starting
Close: Starting => Initial
Up: Initial => Closed
Open: Closed => ReqSent

```
RCR+:   ReqSent => AckSent
TO-:    AckSent => Stopped
RXJ-:   Stopped => Stopped
Down:   Stopped => Starting
Close:  Starting => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
TO-:    AckSent => Stopped
RXR:    Stopped => Stopped
Down:   Stopped => Starting
Close:  Starting => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
Up:     Closing => Closing
Down:   Closing => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
Down:   Closing => Initial
Down:   Initial => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
Open:   Closing => Stopping
Down:   Stopping => Starting
Close:  Starting => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
Close:  Closing => Closing
Down:   Closing => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
TO+:    Closing => Closing
Down:   Closing => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
```

```
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
TO-:    Closing => Closed
Down:   Closed => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
RCR+:   Closing => Closing
Down:   Closing => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
RCR-:   Closing => Closing
Down:   Closing => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
RCA:    Closing => Closing
Down:   Closing => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
RCN:    Closing => Closing
Down:   Closing => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
RTR:    Closing => Closing
Down:   Closing => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
RCR+:   ReqSent => AckSent
Close:  AckSent => Closing
RTA:    Closing => Closed
Down:   Closed => Initial
Close:  Initial => Initial
Up:     Initial => Closed
Open:   Closed => ReqSent
```