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

MATHEUS VOGEL PINTO

Implementação do Protocolo CAN utilizando Simulink para geração automática de VHDL

Trabalho de Graduação.

Prof. Dr. Luigi Carro
Orientador

Porto Alegre, junho de 2011.

AGRADECIMENTOS

Gostaria de agradecer a minha família, meus pais Paulo Sérgio e Bernardete e meus irmãos Thiago e Gabriel, pelo apoio dados todos estes anos e pela compreensão em momentos difíceis.

Aos meus colegas do colégio, que mesmo se encontrando raramente, ainda parece ontem quando nos encontrávamos todos os dias.

Aos colegas de faculdade, pelo chimarrão durante as aulas, truco nos intervalos com Bruno, Antônio, Lucas e Victor, e pela ajuda em fim de semestres.

Ao grupo PET, pela minha primeira experiência de trabalho na área, e que me permitiu muito aprendizado.

Aos professores, da UFRGS e Kaiserslautern, em particular à Profª. Drª. Taisy Silva Weber, por manterem o programa de intercâmbio com a Alemanha, que me permitiu passar um ano lá.

À toda "gurizada" com quem convivi 1 ano na Alemanha, às viagens e muitas outras atividades com Rodrigo Cirando, Henrique Valer e tantos outros, brasileiros ou não. Aos longos cafezinhos depois do almoço com Guilherme Fachini e Carmela Grando, com eventuais participações especiais.

Ao Fraunhofer IESE, por ter me proporciona a realização deste trabalho em parceria com eles.

E a todos mais que participaram direta ou indiretamente deste acontecimento.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ASIC	Application-specific integrated circuit
FPGA	Field-programmable gate array
CRC	Cyclic redundancy check
UFRGS	Universidade Federal do Rio Grande do Sul
VHDL	VHSIC hardware description language
VHSIC	Very-high-speed integrated circuits
HDL	Hardware description language
CAN	Control area network

LISTA DE FIGURAS

RESUMO

Este trabalho apresentará a implementação do protocolo CAN, altamente usado em veículos e equipamentos médicos, em um alto nível de abstração utilizando MATLAB e Simulink, para geração automática de código em VHDL. Além de exigir um período menor de desenvolvimento e menos suscetividade a erros, com ferramentas baseada em modelos, é possível gerar códigos em diferentes linguagens, que possuiriam o mesmo comportamento e utilizando o mesmo modelo. Conseguindo gerar um HDL para esse protocolo, seria possível em apenas um ASIC ou FPGA, ter toda a aplicação e o controlador do protocolo em um componente, sem a necessidade de componentes extras, como um só para aplicação e outro só para o controlador. Será ainda implementado mais um módulo que permite a conexão de vários módulos ao mesmo controlador de comunicação.

Palavras-Chave: CAN, Simulink, VHDL, Geração automática de código.

ABSTRACT

This manual has the purpose of present an implementation of the CAN protocol, highly used in vehicles and medical equipments, in a high level of abstraction using MATLAB and Simulink, to generate automatically VHDL code. Beyond demands a shorter development period and less susceptible to errors, with model-based tools is possible to generate code to different languages, which has the same behavior and using the same model. Generating a HDL code to this protocol, would be possible have in an ASIC or FPGA, the whole application and the protocol controller in one single device, without the need for extra devices, like one for the application and other to the controller. Will be implemented also one more module, that handles the connection of the controller with several application modules.

Keywords: CAN, Simulink, VHDL, Automatic code generation.

1 INTRODUÇÃO

Já se passou o tempo em sistemas embarcados eram apenas micro controladores simples com pouca RAM ou hardwares pequenos, e passaram a ser componentes ou sistemas que executam tarefas complexas e cada vez mais demanda novas aplicações e usos, como DVD Players, smartphones, sistemas de controles automotivos. E esses sistemas muitas vezes são compostos de várias unidades de processamento, que devem cooperar para atingir um objetivo, e ainda, em certos casos, suprir os requisitos de tempo real. Por exemplo, em carros comuns é possível encontrar em torno de 50 dispositivos que controlam diferentes funções, desde o controle de freios e injeção eletrônica até equipamentos de mídia e entretenimento (WEHRMEISTER, 2009).

Devido à evolução tecnológica destes dispositivos e a demanda por novas funcionalidades, desenvolvedores devem se adaptar cada vez mais freqüentemente a novas plataformas de trabalho e desenvolvimento de novas aplicações. Este tempo reduzido e o aumento da complexidade, causam um aumento na chance de se cometerem erros, comprometendo o tempo de desenvolvimento e a qualidade do produto final. Devido a isso, são exigido métodos eficientes de desenvolvimento, como por exemplo, automação de tarefas e reuso de bloco já previamente desenvolvido e validados (WEHRMEISTER, 2009).

Sistemas embarcados, ou de controle, modernos são complexos, e para domínios específicos de aplicação, geralmente possuem subsistemas e interações semelhantes. As similaridades entre esses sistemas tornam a criação de um sistema genérico de desenvolvimento guiado a modelos muito atrativo, permitindo uma fácil adaptação do sistema a uma nova plataforma, ou a criação de um novo utilizando componentes já desenvolvidos. Com as adaptações, esses modelos podem ser executados e verificados em ambientes de simulação, e transformado em software/hardware manual ou automaticamente (SELIC, 2003).

O reuso de componentes em hw/sw já é uma solução parcial para os custos e tempos no desenvolvimento de sistemas complexos. Mas isto já causou várias perdas, como por exemplo, a NASA, que perdeu bilhões devido ao reuso de software com adaptações mal-sucedidas. Esses casos mal-sucedidos talvez se devam ao fato do nível em que essa técnica é utilizada. Geralmente esse reuso em nível de código é bem problemático, mas que poderia ser mais efetivo e seguro em fases anteriores de desenvolvimento num ambiente de desenvolvimento guiado a modelos (WEISS, 2004).

Com um desenvolvimento guiado a modelos, a especificação é estruturada como uma hierarquia de modelos e sua interação entre estes. Isto facilita a definição de requisitos e restrição em nível de sistemas ate os detalhes de design e implementação, auxilia na garantia de propriedades de sistemas e reduz custos de mudança de implementação e revalidação do sistema.

Outra vantagem de uma especificação guiada a modelos é que modelos são expressos com conceitos menos limitantes à tecnologia alvo e mais próximos ao domínio dos problemas do que as linguagens de programação mais populares. Isto faz os modelos serem mais fáceis de serem especificados, entendidos e mantidos, e em certos casos podem permitir que a produção do sistema fique a cargo de um expert da area do problema, ao invés de alguém com conhecimento sobre as tecnologias. E também é menos sensíveis a tecnologia alvo, permite fácil portabilidade para novas plataformas ou versões.

Modelagem de software e geração automática de código já foi tentadas antes, mas com sucesso apenas em áreas específicas. Mas agora temos um melhor entendimento na modelagem de sistemas, esta técnica de desenvolvimento se torna mais útil que no passado, devido a que as ferramentas de automação evoluíram, assim como surgiram padrões no ambiente industrial. Assim a linguagem do modelo toma o papel da implementação das linguagens de programação, assim como elas fizeram com as linguagens assembly (SELIC, 2003).

Neste contexto, a Mathworks possui o Simulink, uma ferramenta de simulação e desenvolvimento guiado a modelos para sistemas embarcados e dinâmicos. Possui um ambiente gráfico e uma série de blocos parametrizáveis para ajudar no desenvolvimento do sistema, assim como add-ons, que possuem já soluções parciais para certos domínios ou que permitam a implementação para plataformas alvo e geração automática de código. Alguns desses add-ons para geração automática de código são, por exemplo, o Real Time Workshop, que gera código C para sistemas de tempo-real ou não, e o HDL Coder que gera código VHDL.

Este trabalho apresenta a implementação de um protocolo de comunicação, mostrando que a partir de um modelo, pode-se gerar automaticamente código, tão confiáveis e quase tão eficiente quanto gerados manualmente.

Este trabalho esta organizado do seguinte modo: O capítulo 2 revisa conceitos básicos de máquinas de estados, assim como algumas definições que as fazem diferentes umas das outras e sobre a ferramenta Stateflow. O capítulo 3 fala sobre geração automática de código e a descreve a ferramenta HDL Coder. No capítulo 4 descreve-se o protocolo CAN, que será o alvo das técnicas e ferramentas previamente descritas. No capítulo 5 será descrita a implementação. No capítulo 6 serão mostrado resultados obtidos dela e uma comparação com um código VHDL gerado manualmente.

2 MÁQUINAS DE ESTADOS

Este capítulo tem por objetivo revisar conceitos básicos de máquinas de estados, e algumas definições, como as conhecidas de Mealy e Moore, e também a de Harel, que foi usada como base para uma das ferramentas utilizadas, o Stateflow, e que não foi vista durante a graduação.

2.1 Conceitos gerais

Uma máquina de estados finita é um modelo comportamental, baseado em estados e transições. Composta por um número finito de estados, em que o estado ativo muda, de acordo com as entradas, que também possuem um número de símbolos finitos, as transições e condições pré-definidas, e que resultam em ações.

Máquinas de estados têm como origem autômatos finitos. Autômatos finitos eram utilizados para reconhecer linguagens. Com uma entrada que satisfizesse a linguagem definida, o autômato que representa essa linguagem deveria chegar a estados finais definidos, e para todas outras entradas não pertencentes à entrada, não. Uma importante aplicação para isto é o uso para verificação de expressões regulares.

Uma máquina de estados finita é definida:

- Por ter um conjunto finito e não-vazio de estados;
- Um estado inicial, no caso de uma máquina determinística. No caso de uma máquina não-determinística, um conjunto de estados;
- Um conjunto finito de entradas;
- Um conjunto finito de saídas;
- Uma função de transição dos estados em função do estados atual e das entradas;
- Uma função de saída.

Estas são as definições básicas de uma máquina de estados, mas que deixam espaço para a formulação de máquinas com características diferentes. Serão mencionado duas das mais conhecidas variações, e na subseção que fala sobre o Stateflow, será explicado algumas características da definição de Harel para máquinas de estados presentes na ferramenta.

Mesmo com estas diferenças, suas modelagens geralmente são feitas usando-se tabelas de transição de estados ou diagramas.

Existem variações nas técnicas de modelagem usando-se tabelas de transição. Podem ser de uma dimensão, o que as faz muito similares a uma tabela verdade. Possuem colunas para as entradas e estado atual, e em seguida para o próximo estado e eventualmente para saídas, dependendo a definição da máquina de estado que está sendo usada (com os eventos sendo gerados nas transições ou decorrente do estado atual). Podem ser modeladas também com tabelas bi-dimensionais, tendo como uma dimensão o estado atual, e a outra o próximo estado ou o evento de entrada. No caso da dimensão ser o próximo estado, a célula possui a condição para que a transição ocorra, e caso a dimensão seja o evento de entrada, a célula possui o próximo estado. Em ambos, ainda existe a possibilidade de haver mais um campo, que seria o evento a ser propagado pela transição.

Máquinas de estados também podem ser modeladas por diagramas, onde estados são representados. Esta é apenas mais uma maneira de modelar máquinas de estados, e as suas variações dependem também da representação de máquina de estados que será usada.

2.2 Mealy

Máquinas de estados de Mealy se caracterizam pela suas saídas serem em função do estado atual e das entradas. Em um diagrama de Mealy, podemos associar as saídas com as transições. Mas devido a isso, as saídas podem mudar assincronamente com o clock, causando uma dificuldade maior para análise temporal.

Porém, em relação a uma máquina do tipo Moore, há uma grande simplificação, já que necessita de um número menor de estados. Por exemplo, em Mealy, há um estado que possui três transições que tem o mesmo estado destino, mas ações diferentes, em Moore, seriam necessário três estados destinos diferentes.

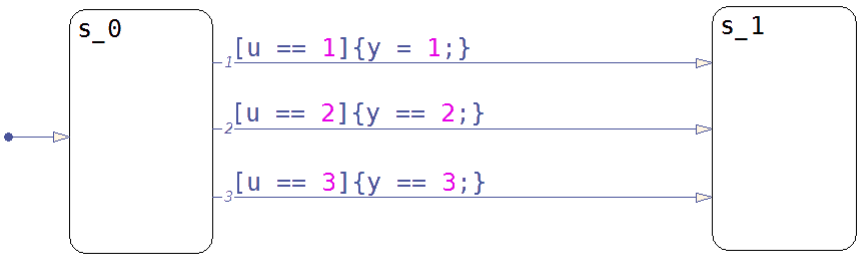

Figura 2.1: Exemplo de uma máquina de estados de Mealy

2.3 Moore

Em máquinas Moore, as saídas são definidas apenas em função do estado atual, o que ocasiona que as saídas se modificam sincronamente com o estado atual. Assim como já foi explicado na seção anterior, máquinas Moore necessitam de um número maior de estados, um para cada situação do sistema modelado.

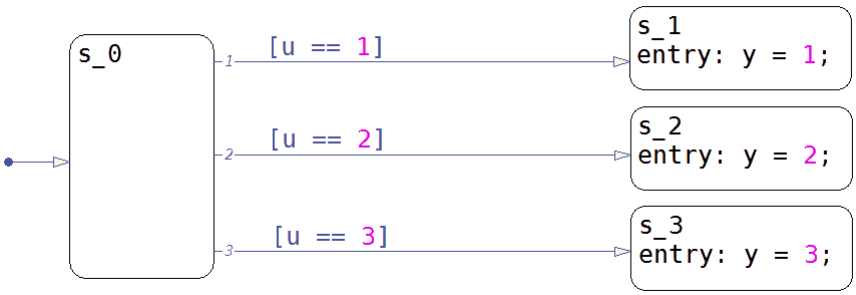

Figura 2.2: Exemplo de uma máquina de estados de Moore

2.4 Stateflow

O Stateflow é um componente presente no Similink, a linguagem gráfica presente no Matlab. Uma máquina de estados do Stateflow é um tipo de bloco em um modelo do Simulink. Executando a simulação, executa tanto blocos do Simulink, como gráficos do Stateflow.

Uma máquina de estados do Stateflow é composta por objetos gráficos como estados, funções, notas e junções. Cada máquina de estados no modelo aparece como um único bloco, cada um tendo sua própria hierarquia, que é composta por objetos gráficos e não-gráficos.

Uma máquina de estados do Stateflow utiliza uma variante da máquina de estados definida por Harel, que pode ser visto um exemplo na figura 2.3.

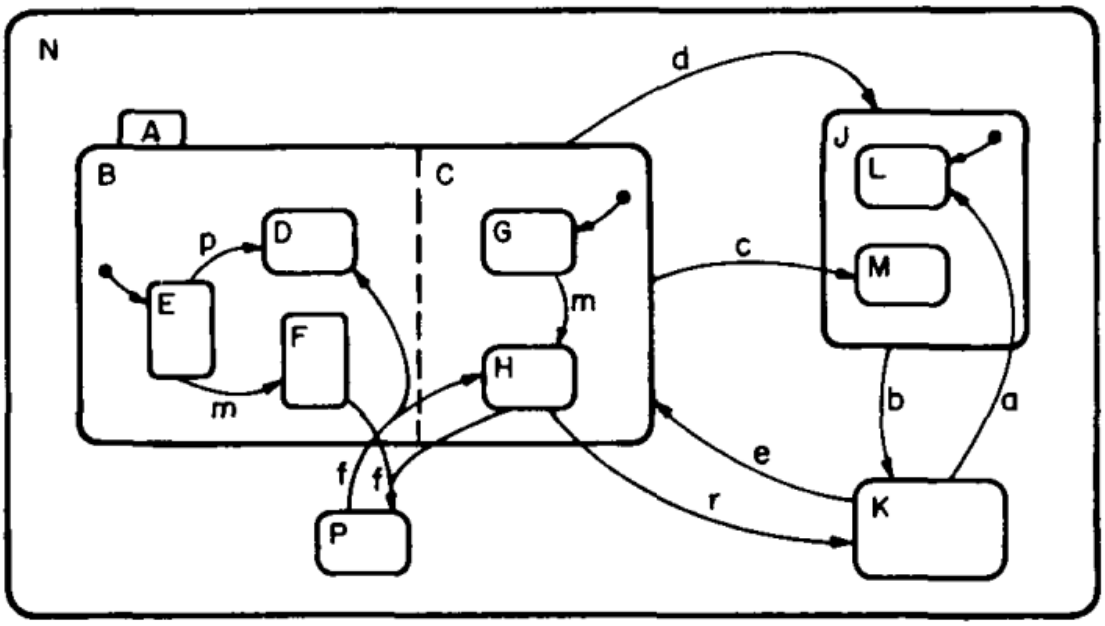

Figura 2.3: Exemplo de uma máquina de estados com a definição de Harel

Um gráfico é uma representação de uma máquina de estados finita, onde estados e transições formam a base da estrutura. Também podem ser representadas gráficos sem estados. Um gráfico do Stateflow possui representações para utilização de hierarquia, paralelismo e história. Com isso é possível criar gráficos com sub-estados dentro de super estados, múltiplos estados ativos ao mesmo tempo e transições baseadas em históricos de informação. Também é possível adicionar funções do Matlab ao diagrama.

Na figura 2.4 podem-se observar quase todos os recursos oferecidos para a modelagem de um problema.

- State with parallel (AND) decomposition: permite que se modelem máquinas com estados paralelos. Os estados com linhas pontilhadas são paralelos entre si, e dentro deles é possível que se adicione sub-estados e transições, fazendo que se tenham duas máquinas ativas em execução;
- Superstate e substate: é possível fazer aninhamento de estados. Isto permite uma maior organização, e também agrupar um número de estados semelhantes, que, por exemplo, precisem executar uma mesma tarefa, ou que tenham as mesmas condições de transição para um mesmo estado ou conjunto deles;
- Default transition: serve para definir o estados inicial do sistema, ou de sub-estados de super-estados que se tornam ativos;
- Connective junction: ajuda na representação em que um estado pode ter múltiplos destinos, múltiplas origens terem um destino, e para fazer estruturas do tipo if-then-else, para calcular as transições as serem tomadas;
- State actions: são ações que são tomadas sempre que ocorrem os eventos definidos por ela. Podem ser em entradas ou saídas do estado, ou em eventos configurados no Stateflow;
- Condition e Condition action: as condições são o que definem as transições que o sistema irá executar, e a elas é possível gerar ações quando elas ocorrem;
- Flow graph: é um conjunto de junções, transições, condições e ações que facilitam o calculo das saídas e próximo estado.

Pode-se ver também que no caso de junções com múltiplas saídas ou estados paralelos, há números que marcam alguns objetos. No caso das junções com múltiplas saídas, isso é a ordem que as condições são testadas. Caso uma não seja verdadeira, será verificada a próxima. Mas com estados paralelos, é a ordem com que cada máquina é executada. Formalmente são estados paralelos, mas elas não executam concorrentemente. Cada um tem uma prioridade, e é executado uma depois da outra, mas ambas estão ativas e executa no mesmo ciclo.

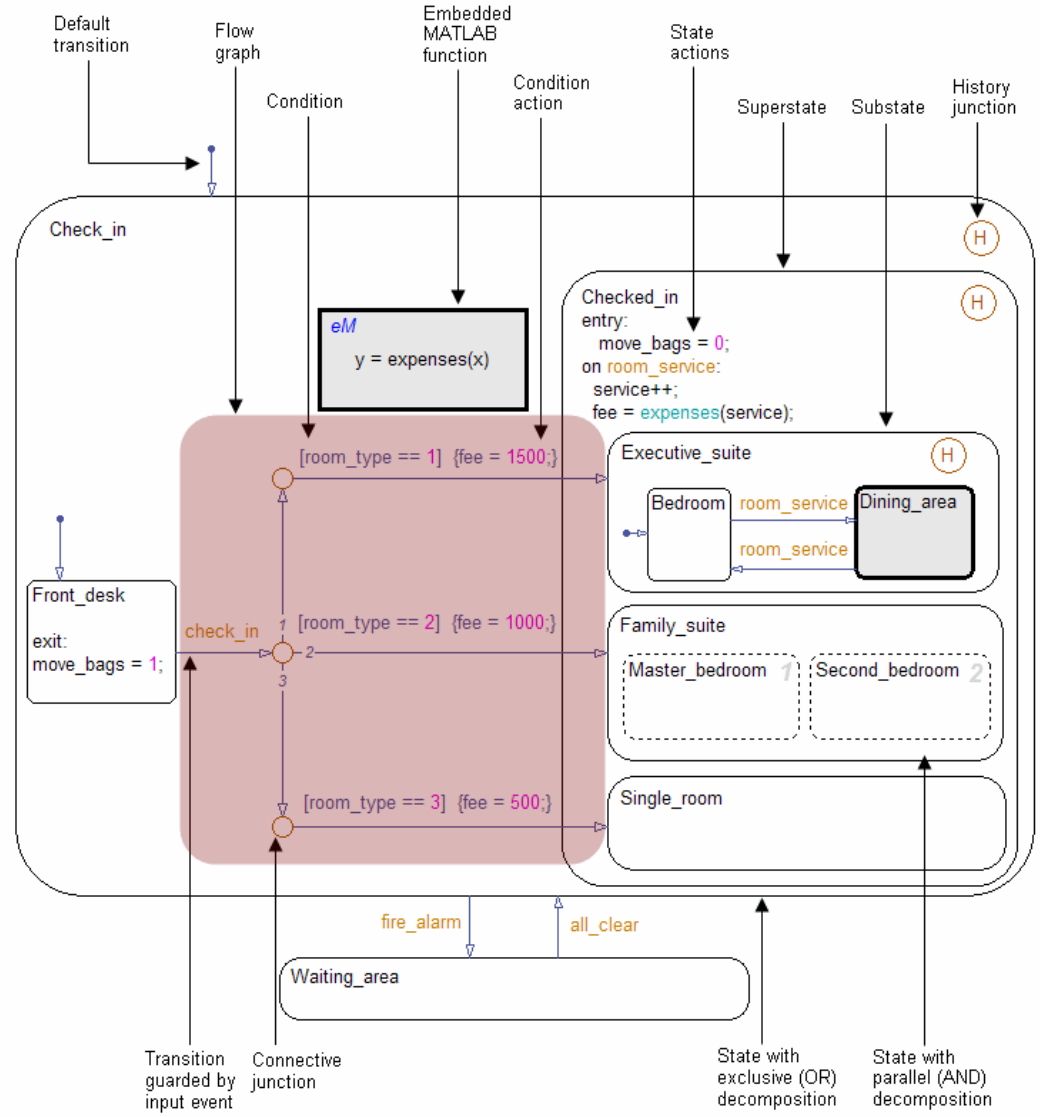

Figura 2.4: Exemplo de um diagrama de máquina de estados do Stateflow

2.5 Em hardware

Na prática, uma máquina de estados, quando sintetizada, pode ser considerada como tendo a configurações das figura 2.5 ou 2.6. Uma lógica que calcula o próximo estado, um registrador que armazena o estado atual, e uma lógica gera as saídas do sistema em função do estado atual, e no caso de Mealy, também em função das entradas.

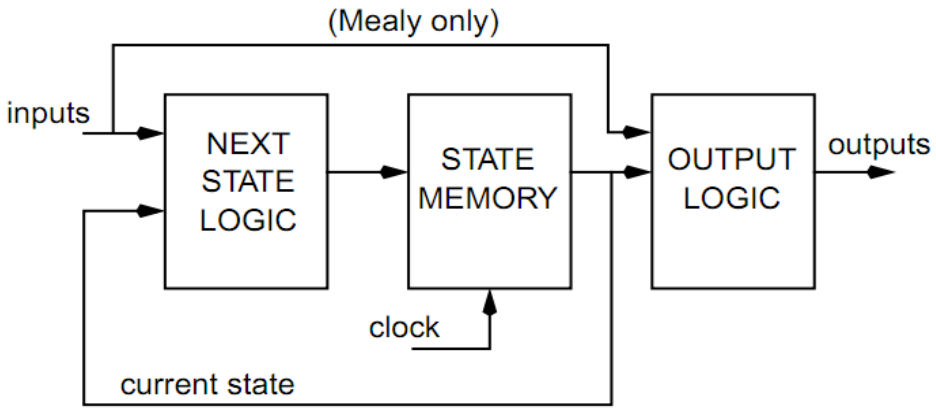

Figura 2.5: Um modelo da implementação de máquina de estados em hardware

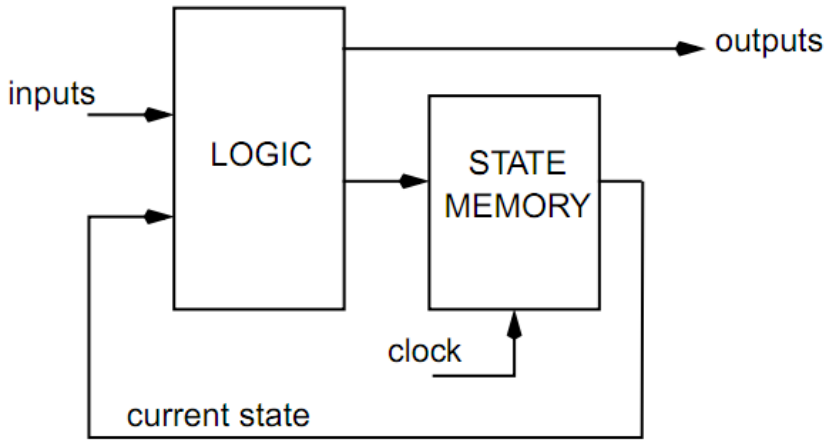

Figura 2.6: Um modelo alternativo para a implementação de máquina de estados em hardware

3 GERAÇÃO AUTOMÁTICA DE CÓDIGO

Foi utilizado a ferramenta Simulink, pois é uma ferramenta de muita utilização nos meios de engenharia, e é o ambiente mais utilizado para modelagem e especificação na área de controle embarcado e modelagem de sistemas. Também por que se tinha licença para os recursos necessários, o que permitiria manter todo fluxo de trabalho em apenas uma ferramenta, da modelagem até a geração de código.

3.1 Simulink

O Simulink é uma ferramenta para modelagem e simulações de sistemas, como processamento de sinais, e até simulações de sistemas aniônicos. Por já possuir uma biblioteca vasta de recursos e extensões para as mais diversas aplicações, é uma ferramenta que facilita o desenvolvimento de aplicações.

Mas geralmente é apenas utilizada para a modelagem do sistema e teste dos parâmetros, sendo os conceitos ou sistemas modelados ser reimplementados para uma plataforma alvo, um programa em C, um circuito para FPGA ou ASIC em VHDL ou filtros para circuitos analógico são exemplos. Mas o Simulink possui extensões que é possível gerar código automaticamente a partir de seus modelos. Feitas pela própria Mathworks para o Matlab, pode-se comprar licença, ou adquirir versões acadêmicas em algumas delas.

Mas como seus modelos são genéricos e primariamente criados serem utilizados dentro de suas plataformas, a geração de código para estas ferramentas possui suas limitações, que se diferenciam dependendo a linguagem desejada.

O Stateflow, como extensão do Simulink, também herda esta característica, de geração de código, também limitações, não se podendo, até o momento, utilizar todos os recursos dele disponíveis, mas que a cada versão agrega novas capacidades.

3.2 Simulink HDL Coder e suas limitações

O Simulink HDL Coder é um software que permite a geração de código de descrição de hardware baseados em modelos do Simulink e máquina finita de estados do Stateflow. Este gerador traz a abordagem do design baseado em modelos para o domínio do desenvolvimento de ASICs e FPGAs. Com está ferramenta os designers podem passar mais tempo refinando seus algoritmos e prototipando seus sistemas e menos tempo na codificação.

Os principais recursos citados pela ferramenta são:

- Geração de HDL sintetízavel independente de plataforma, a partir de modelos do Simulink, código do Matlab e diagramas do Stateflow;
- Suporte a máquinas de estados de Mealy e Moore e implementações de lógicas de controle;
- Geração de testbenches e co-simulação de modelos com EDA Simulator Link;
- Opções de compartilhamento de recursos e retiming de sub-sistemas para trade-off de área-velocidade;
- Otimização do modelo do Simulink usando informações de restrições de tempo e de ferramentas de síntese de HDL;
- Integração de código legado.

Apesar de automatizar a geração de código e facilitar o desenvolvimento do hardware, já é esperado de seus usuários prévios conhecimento em HDL, desenvolvimento de hardware e outros conhecimentos que seriam necessários para se executar o tradicional de design de sistemas digitais.

Mas, como já foi mencionado, a geração de código limita a vasta gama de modelagem que o Simulink proporciona, já que por muitas vezes a geração de código de blocos complexos não seriam uma tarefa fácil, ou talvez até possível, para VHDL.

Quanto a essa limitação de modelagem, o HDL Coder já possui uma referência de quais blocos presentes no Simulink, que ele possui suporte, e mais alguns novos, que são bastante úteis no desenvolvimento de sistemas digitais e não estão presentes no Simulink.

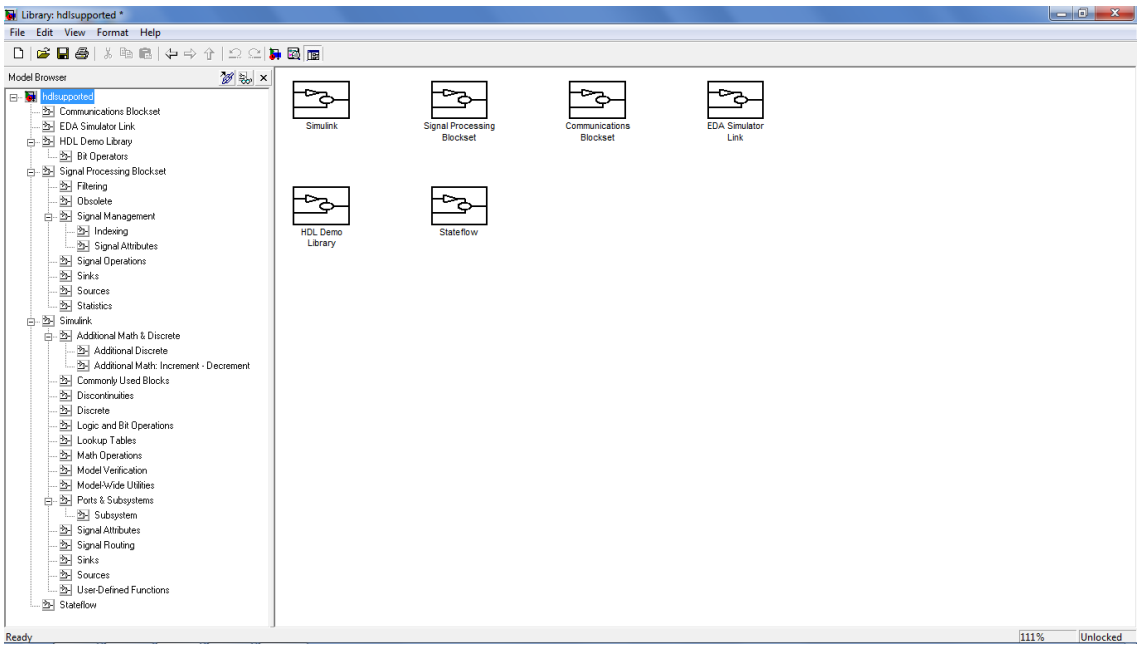

Figura 3.1: Janela com as bibliotecas e blocos suportados pelo HDL Coder

Utilizando o Stateflow, pode-se gerar uma máquina de estados, ou um algoritmo de controle complexo. Em geral, a geração de HDL de um modelo contendo uma máquina de estados, não se difere muito da geração para outro modelo qualquer. O HDL Coder foi desenvolvido para:

- Dar suporte ao máximo número de recursos que o Stateflow possui que são consistentes com HDLs. Isto permite que se gere código HDL a partir de modelos já existentes sem mudanças significativas;
- Gerar código HDL bit-true e cicle-accurate que é totalmente compatível com a semântica de simulação do Stateflow.
- Agora serão citada algumas condições para a geração de código a partir do Stateflow que tiveram que ser respeitadas para a implementação deste trabalho, assim como alguns recursos providos por eles:
- Registered Output: esta opção permite que as saídas do diagrama mantenham seus valores entre um ciclo e outro, se não são modificados;
- Não utilizar geração de eventos locais no diagrama. Não usar os eventos enter, exit e change. wakeup e tick são permitidos;
- HDL não suporta goto. E também não fazer diagramas de fluxo não-estruturados, isto quer dizer, que não seja possível reproduzir com estruturas if-then-else, como por exemplo, na figura 3.2.

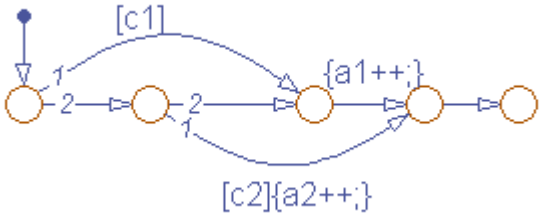

Figura 3.2: Exemplo de diagrama de fluxo não-estruturado

4 APLICAÇÃO: PROTOCOLO CAN

O protocolo CAN é um protocolo de comunicação serial que suporta eficientemente controle distribuídos de tempo real com um alto nível de segurança. Seu domínio de aplicação vai de redes de alta velocidade a multiplexação de cabeamentos de baixo custo. Em eletrônicos de veículos automotivos, em unidades de controle de mecanismos, sensores, etc. são conectados usando o protocolo CAN em velocidades de até 1Mbps.

4.1 Características gerais

As propriedades básicas que o protocolo segue são:

- Priorização de mensagens;
- Garantia do tempo de latência;
- Flexibilidade de configuração;
- Recepção multicast com sincronização de tempo;
- Consistência de dados no sistema;
- Multi-mestre;
- Detecção e sinalização de erros;
- Retransmissão de mensagens corrompidas assim que o bus volta ao estado de idle;
- Distinção entre erros temporários e falhas permanentes do nodo e desligamento autônomo de nodos defeituosos.

O CAN também possui uma organização arquitetural de acordo com o modelo de referência OSI:

- A camada física define como os sinais são realmente transmitidos, e lidam com as descrições de tempo de bit, codificação do bit e sincronização. Com a especificação usada, e também o escopo do trabalho, o driver e a codificação no meio elétrico não são definidas, o que permite que a implementação em níveis de sinal sejam otimizadas para a aplicação alvo.
- A camada MAC representa o núcleo do protocolo. Esta camada é responsável por “message framing”, arbitração, acknowledgement, detecção de erros e sinalização. Esta camada é supervisionada por uma entidade de confinamento de erros;

- A camada LLC é responsável pela filtragem de mensagens, notificação de sobre carga e gerenciamento de recuperação.

As mensagens são enviadas em formatos fixos de tamanhos variáveis, mas limitados. Quando o barramento está livre, qualquer nodo pode começar uma transmissão.

Em um sistema CAN, nenhum nodo faz uso de qualquer informação sobre a configuração do sistema, e isso ocasiona em importantes conseqüências:

- Flexibilidade do sistema: nodos podem ser adicionados no sistema sem ser necessárias modificações de software ou hardware nos outros nodos;
- Roteamento da mensagem: o conteúdo da mensagem é definido com um identificador. O identificador não indica o destino da mensagem, mas o significado do pacote, então está a cargo de cada nodo filtrar a mensagem e decidir se o dado deve ser tratado ou não;
- Multicast: com isso e o filtro de mensagens de cada nodo, qualquer número de nodos podem receber e agir simultaneamente com a mesma mensagem;
- Consistência de dados: uma mensagem é aceita por todos ou nenhum nodo.

Outras características do protocolo são:

- O protocolo não define velocidade obrigatória na utilização em sistemas. Apenas que dentro de um mesmo sistema, seja uniforme e fixa.
- O identificador define uma prioridade estática da mensagem durante o acesso ao barramento;
- Requisição remota de dados: enviando um “remote frame” um nodo que necessita um dado pode requisitá-lo no barramento;
- Quando o barramento estiver livre, qualquer nodo pode começar a transmitir sua mensagem;
- Arbitração: quando o barramento estiver livre, qualquer nodo pode começar uma transmissão. Se duas mensagens começarem a transmissão ao mesmo tempo, o conflito pode ser resolvido utilizando uma comparação bit a bit entre os identificadores. O mecanismo garante que nenhuma informação ou tempo são perdidos. Durante a arbitração, todos transmissores comparam o bit transmitido como o lido no barramento. O barramento possui os níveis dominante e recessivo. Sempre que dois transmissores escreverem algo no barramento, o bit dominante é o que permanece no barramento. Quando a arbitração é perdida, o transmissor que perdeu passa a ser receptor. Um exemplo está na figura 5.x.
- Possui cinco mecanismos de detecção de erros: CRC, Bit stuffing, monitoramento (no caso de transmissores que devem comparar o nível escrito com o lido no barramento) e Message Frame Check e acknowledgement.

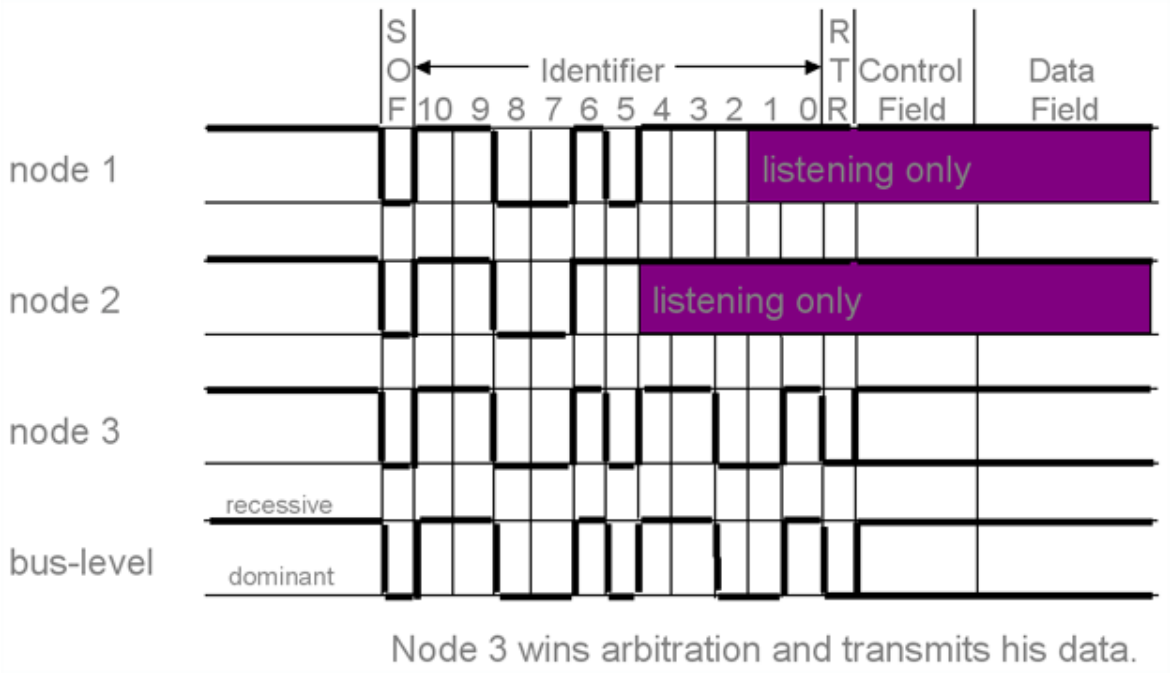

Figura 4.1: Exemplo de escolha do transmissor por arbitração

4.2 Tipo de frames

4.2.1 Data ou Remote frames

Consiste dos seguintes segmentos:

- SOF (start of frame):
  - Delimita o começo de uma mensagem e consiste de um bit dominante;
- Arbitration Field:
  - É o identificador da mensagem e critério de desempate quando duas mensagens começam a transmitir ao mesmo tempo;
  - No formato padrão possui 11 bits de identificador e um bit que o identifica como remote ou data frame;
  - No formato estendido possui 29 bits de identificador, um bit que o identifica como formato extendido, um bit que o identifica como data ou remote frame, e outro bit a fim de fazer os dois formatos serem compatíveis;
- Control field:
  - para um frame do formato padrão: um bit que o identifica como um frame do formato padrão, um bit reservado e quatro bits que representam o tamanho do dado para mensagens deste identificador;
  - para um frame do formato extendido: dois bits reservados e quatro bits que representam o tamanho do dado para mensagens deste identificador;
- Data field:

- para Remote frames, seu tamanho são zero bits, ou seja, inexistente;
- para Data frames varia de 0 a 8 bytes;

- CRC field:
  - possui 15 bits para o CRC, que é calculado do SOF ao fim do data field e 1 de delimitador;

$$X^{15} + X^{14} + X^{10} + X^{8} + X^{7} + X^{4} + X^{3} + 1$$

Figura 4.2: Polinômio gerador do CRC do protocolo

- ACK field:
  - possui 1 bit de para que os receptores transmitam seus ACKs. Neste momento o transmissor escreve um bit recessivo no barramento. Os receptores enviam um bit dominante caso não tenha ocorrido nenhum erro durante a recepção;
  - e um bit para delimitar o campo;
- EOF (end of frame):
  - consiste de 7 bits recessivos.

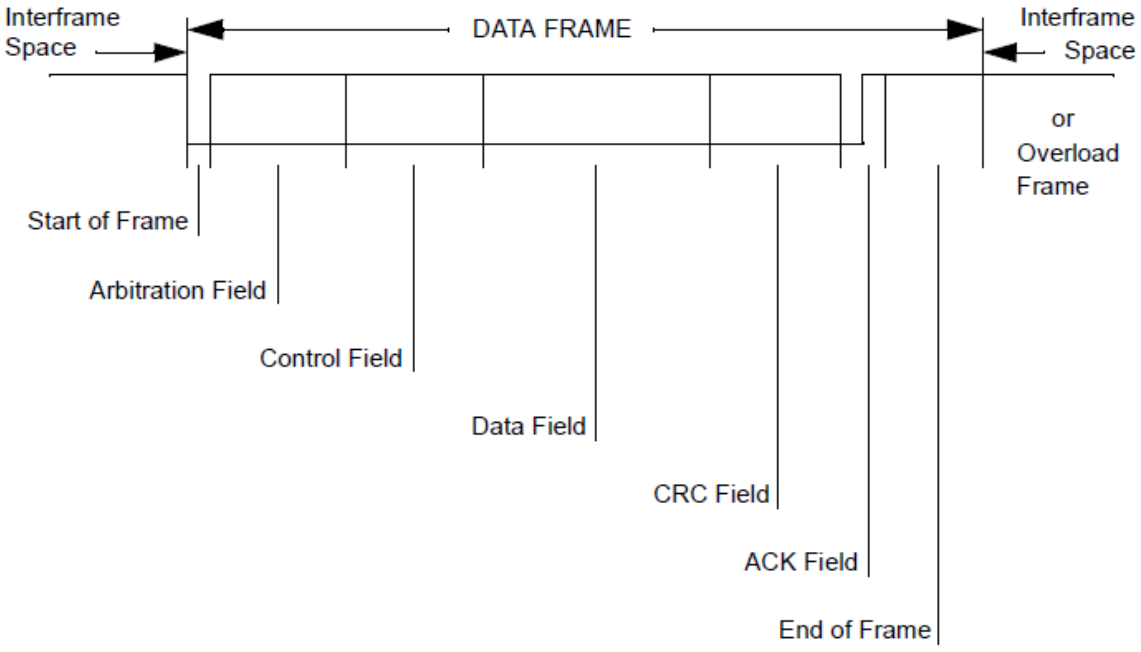

Figura 4.3: Formato de um Data frames padrão

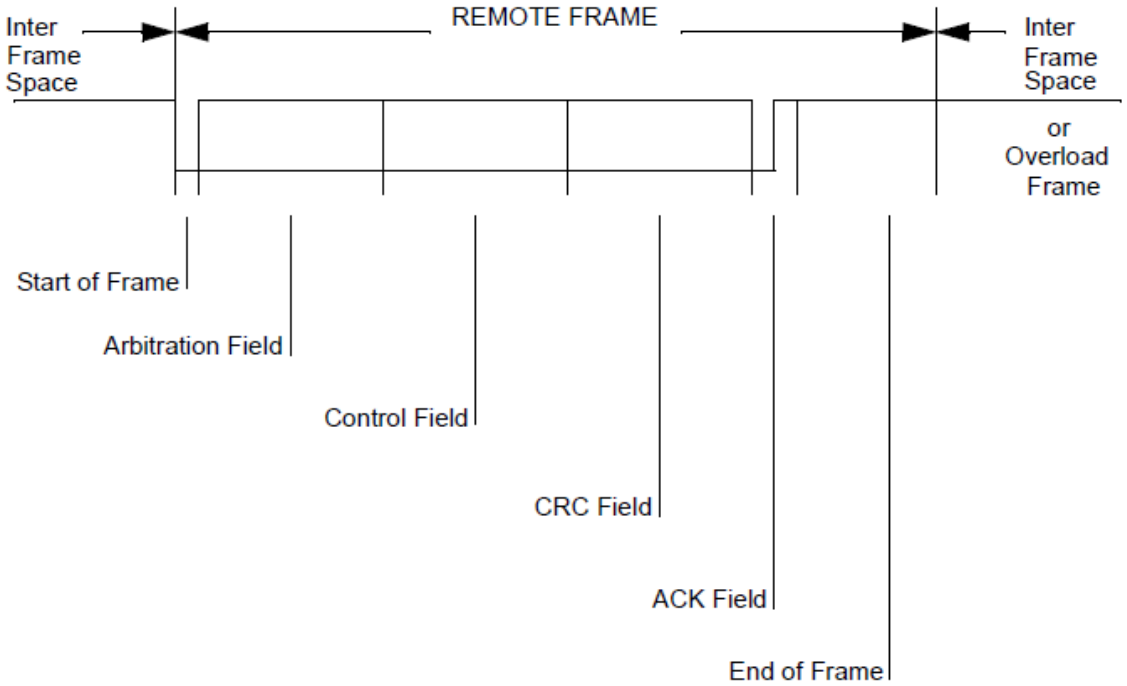

Figura 4.4: Formato de um Remote frame padrão

4.2.2 Error frame

É composto por dois campos chamados error flag e error delimeter.

Um campo de error flag pode ser ativo ou passivo, dependendo de seu status atual, que será discutido mais adiante. Sempre que um nodo detectar uma condição de erro, ela é enviada. Ela destrói a regra do bit stuffing e os formatos fixos do ACK field e EOF. Como conseqüência, se os outros nodos não detectaram erro, eles o detectarão agora, e enviaram a error flag também. A flag é composta por 6 bits dominantes, mas este campo pode se extender a 12 bits dominantes, devido à superposição da flag entre vários nodos.

O error delimeter consiste de 8 bits recessivos seguidos, que começam após o último bit da error flag serem recebido.

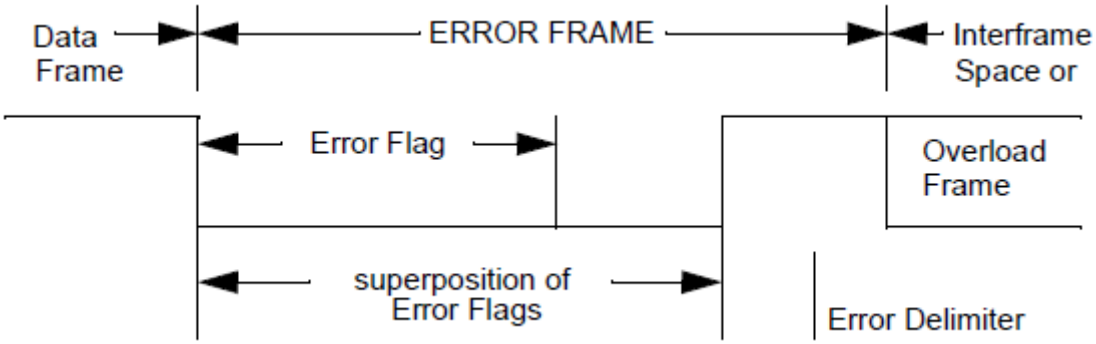

Figura 4.5: Formato de um Error frame

4.2.3 Overload frame

Possui o mesmo formato de um error frame, mas é utilizado em casos que o receptor necessita de um atraso extra para próxima transmissão de remote ou data frames. Este frame destrói o formato do intermission.

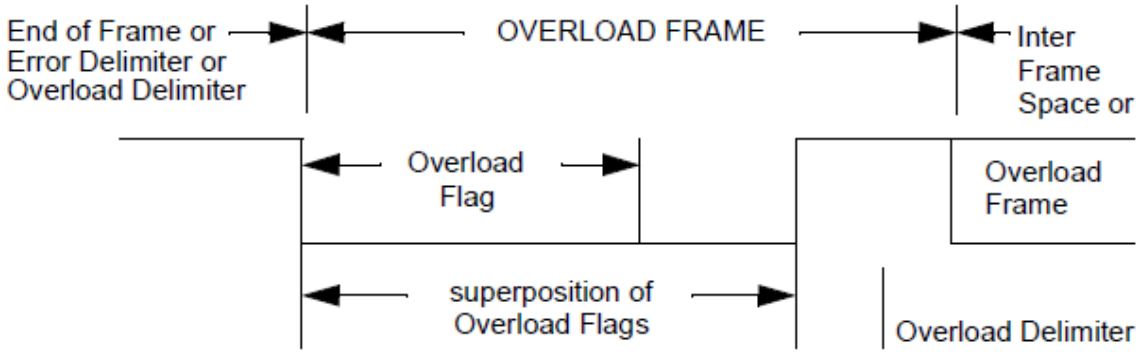

Figura 4.6: Overload Frame

4.2.4 Interframe spacing

Data e Remote frames só podem ser transmitidos após o envio deste frame. Os outros frames não necessitam deste para serem transmitidos.

Para nodos error active, é composto de 3 bits de intermission e um número arbitrário de bits para determinar que o barramento está livre. Durante o intermission pode-se enviar um frame de Overload.

Para nodos error passive, é composto pelo mesmo intermission, mas seguido de 8 bits recessivos caso tenha sido o transmissor da última mensagem.

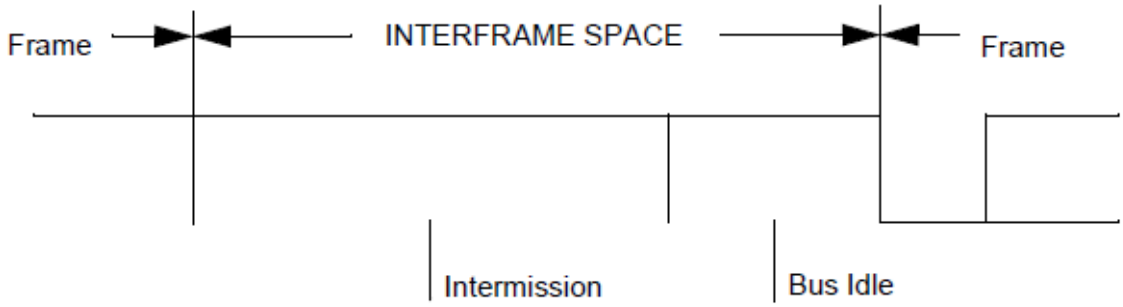

Figura 4.7: Interframe space

4.3 Codificação

A codificação do stream de bits segue as seguintes regras:

- Os segmentos de frames SOF, arbitration field, control field, data field e CRC são codificados pelo método do bit stuffing. Sempre que houver uma seqüência de 5 bit de mesma polaridade, deve ser inserido um de polaridade oposta;
- Os demais seguimentos são fixos e não é utilizados nenhum método de bit stuffing;
- Os bits seguem o padrão de Non-Return-To-Zero, isto significa que durante todo o tempo de bit, o bit é ou dominante ou recessivo.

4.4 Erros

São definidos 5 tipos de erros, que não são mutuamente exclusivos:

- Bit error: um nodo transmissor monitora o valor do barramento. Sempre que for monitorado um valor diferente daquele escrito, será considerado um erro, com exceção quando ocorrerem durante a arbitração, ACK slot, ou durante a transmissão da flag de erro passiva;
- Stuff error: deve ser detectado durante o sexto bit consecutivo de mesma polaridade nos segmentos que é utilizada essa codificação;
- CRC error: ocorre quando o crc calculado no receptor é diferente do recebido do transmissor;
- Form error: ocorre quando bits das partes fixas dos frames se diferem dos valores padrões definidos no protocolo;
- ACK error: ocorre quando o transmissor não detecta um bit dominante durante o ACK slot.

Quando o erro é detectado, nodos "error active" enviam active error flags, e nodos "error passive" enviam passive error flags. Para bit error, stuff error, form error e ACK error deve se começar a enviar a flag de erro já no próximo tempo de bit. CRC error devem ser reportados após o ACK delimeter.

4.5 Confinamento de erros

Os nodos podem estar em um dos seguintes 3 estados: error active, error passive e bus-off. Quando erros são detectados, nodos error active enviam active error flags e nodos error passive enviam passive error flags. Um nodo "bus-off" não deve exercer nenhuma influência sobre o barramento.

Todas as unidades devem conter 2 contadores a fim de definir o estado que eles se encontram um para contar erros de transmissão e outro para contar erros de recepção. Existem um conjunto de regras para definir qual o quanto esses contadores serão incrementados.

Se um dos contadores possuir um valor maior que 128, nodo passa a ser error passive. Um nodo com o contador de transmissão que for maior ou igual a 256 passa a ser bus-off. Estados bus-off ou error passive podem voltar a ser error active dependendo das condições que ocorrerem.

4.6 Tempo de bit

O tempo de 1 bit é dividido em 4 segmentos:

- segmento de sincronização: usado para a sincronização dos nodos. As bordas dos bits são esperadas neste segmento;
- segmento de tempo de propagação: serve para compensar o atraso do meio físico na rede;
- segmentos de fase 1 e 2: usados para compensar o erro de fase entre os nodos. Podem ser extendidos ou encurtados para fins de sincronização.

Um bit é dividido em unidades de tempo. O comprimento dos segmentos são:

- Segmento de sincronização: 1 unidade de tempo;
- segmento de propagação: de 1 a 8 unidades de tempo;
- segmento de fase 1: de 1 a 8 unidades de tempo;

O segmento de fase 2 é o maior valor entre o segmento de fase 1 e o tempo de processamento da informação, e o tempo de processamento da informação deve ser menor ou igual a 2 unidades de tempo;

O tempo de bit deve ter entre 8 e 25 unidades de tempo.

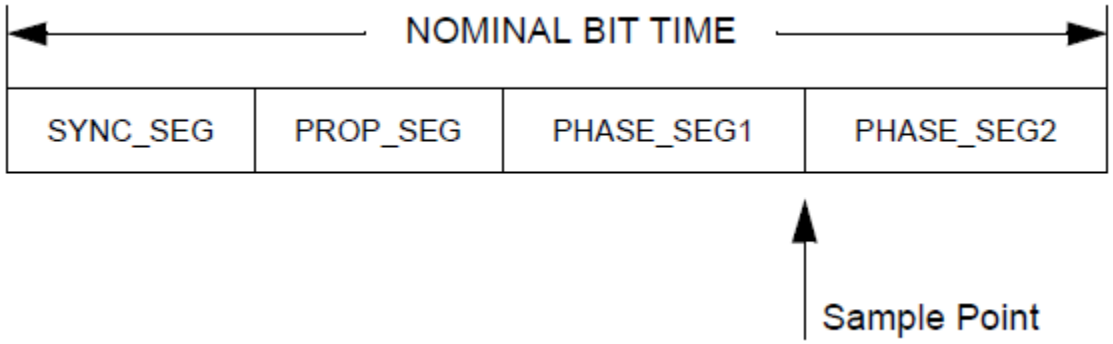

Figura 4.8: Segmentos de bit

4.7 Sincronização

Há dois tipos de sincronização: hard sincronization e resincronization.

4.7.1 Hard Sincronization

No barramento, em uma borda de bit recessivo para dominante, força o segmento de bit atual seja o segmento de sincronização.

4.7.2 Resincronization

Para fins de sincronização, o segmento de fase 1 pode ser extendido, ou o segmento de fase 2 comprimido. O limite máximo desta mudança de tamanho é definido pelo RESYNCHRONIZATION JUMP WIDTH, que é um valor programável entre 1 e o menor valor entre 4 e o comprimento do segmento de fase 1.

O erro de fase é dado pela posição relativa da borda com o segmento de sincronização. O erro é zero quando a borda se situa no segmento de sincronização, negativo quando se ocorre antes, e positivo quando ocorre depois. Quando o erro é positivo o segmento de fase 1 é extendido de acordo com o valor do erro, mas no máximo de RJW. E quando o erro é negativo, o segmento de fase 2 é comprimido.

4.7.3 Regras para a (re-)sincronização

- Apenas uma sincronização por bit é permitida;
- Hard sincronization será usada quando houver uma borda de um bit recessivo para dominante quando o barramento está em idle;

- Nos outros casos de bordas recessivas para dominantes, será utilizado a resincronização.

5 IMPLEMENTAÇÃO E RESULTADOS PRÁTICOS

Agora será descrito a arquitetura projetada para implementação do protocolo CAN, sendo modelado usando-se as ferramentas Simulink e Stateflow para gerar o modelo, e o HDL Coder para geração de código VHDL genérico sintetízavel.

A estrutura geral do módulo está presente na figura 5.1.

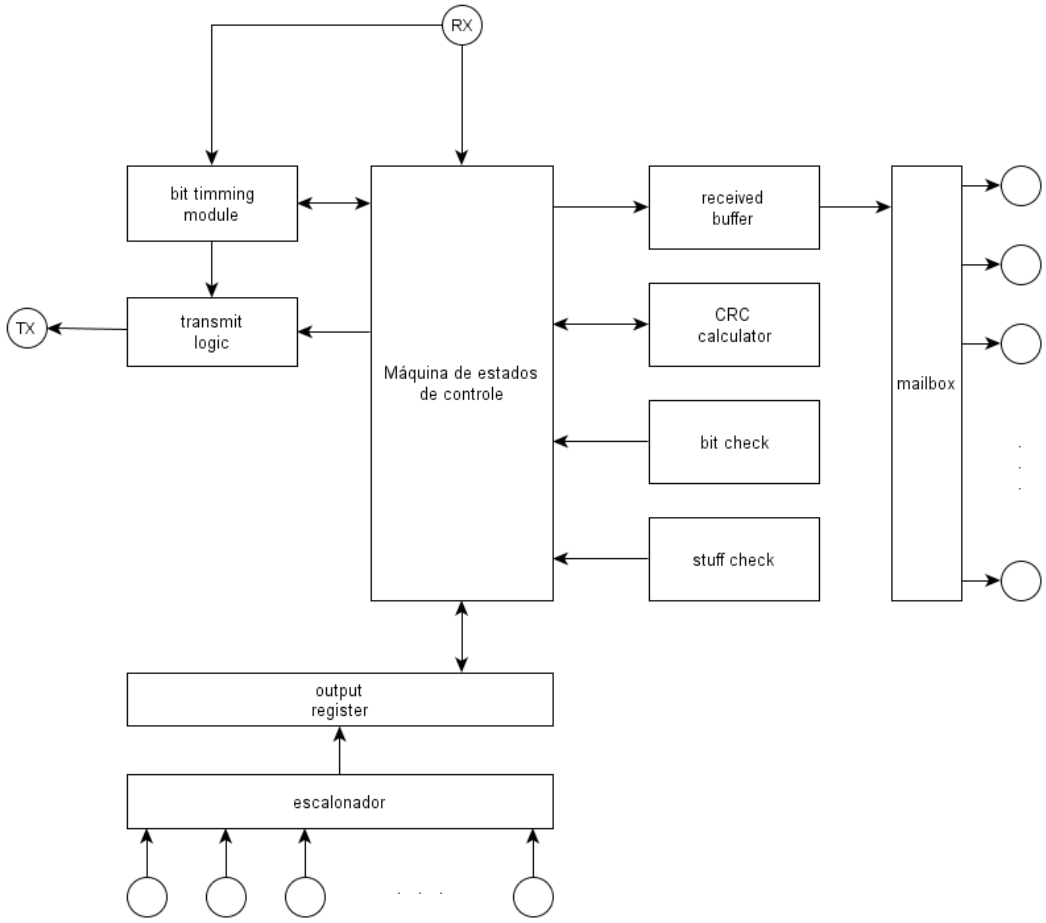

Figura 5.1: Arquitetura do controlador do protocolo CAN

Nas seções seguintes será descrito a característica dos módulos do controlador.

5.1 CRC calculator

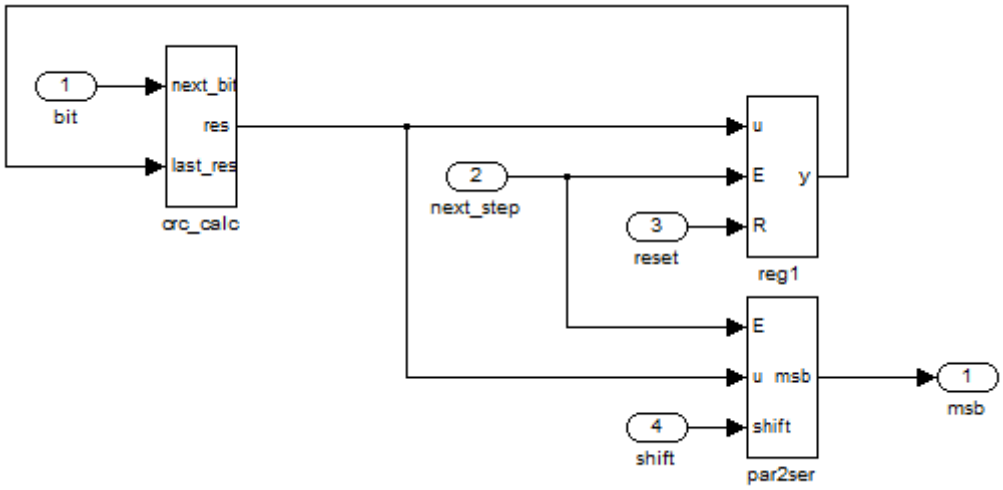

Figura 5.2: Estrutura geral do módulo de cálculo do CRC

O módulo de cálculo do CRC possui as seguintes características:

- next_step é o sinal de controle para que seja calculada a próxima iteração do algoritmo;
- reset reinicia o valor do registrador que armazena o resultado parcial do calculo;
- cada iteração é calculada quando chega um novo bit que faz parte dos segmentos que necessitam do cálculo de CRC;
- há um shift register que é responsável serve para a transmissão do CRC, e também pela comparação com cada bit que seja sabendo já no mesmo ciclo se houve erro no cálculo do CRC;

O módulo implementa o seguinte algoritmo, descrito na especificação:

```
CRC_RG = 0; // initialize shift register
REPEAT
    CRCNXT = NXTBIT EXOR CRC_RG(14);
    CRC_RG(14:1) = CRC_RG(13:0); // shift left by
    CRC_RG(0) = 0; // 1 position
    IF CRCNXT THEN
            CRC_RG(14:0) = CRC_RG(14:0) EXOR (4599hex);
    ENDIF
UNTIL (CRC SEQUENCE starts or there is an ERROR condition)
```

5.2 Bit timming

Este módulo é o responsável pela sincronização e re-sincronização do sistema com a rede. É implementado em uma máquina de estados e é responsável por gerar os sinais que indicam momento de amostragem do bit, que serve como um trigger para a máquina de estados de controle, e também quando se inicia um novo tempo de bit, para a lógica de transmissão.

Toda a implementação desse módulo foi feita utilizando se uma máquina de estados, que como entrada possui o estado do barramento (se está em idle ou não) e um detector de borda de descida, que é onde ocorre os eventos para sincronização.

O tamanho dos segmentos é configurável no modelo do Simulink. A combinação destes segmentos e a freqüência de clock que definirão a taxa de transmissão, ou recepção, do módulo.

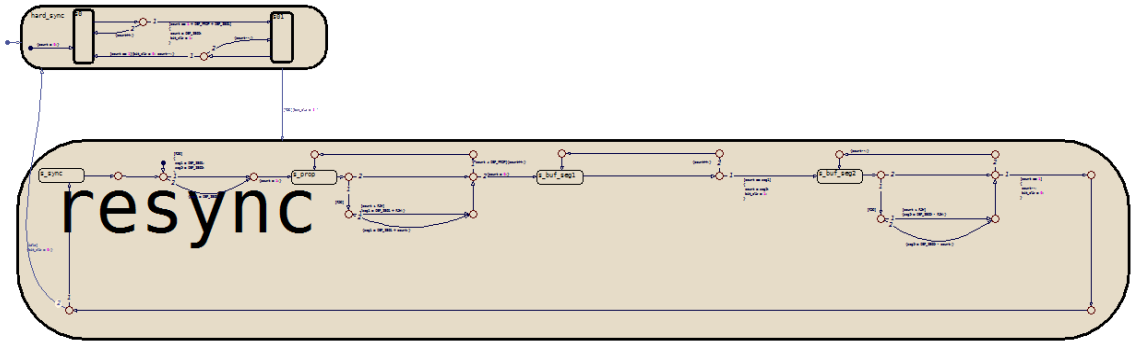

Figura 5.3: Visão geral da implementação do módulo de bit timming

5.3 Mailbox e buffer de recepção

Neste sistema, todos os pacotes são recebidos, inclusive os que são enviados. Quando há a transmissão de um frame, no começo pode haver dois nodos competindo pelo barramento. O nodo com mensagem de menor prioridade pára a transmissão, mas deve ter recebido todos os dados que antes vieram, no caso, a prioridade da mensagem. Após isso foi decidido continuar a recepção de mensagens que o próprio nodo envia, para fins de diminuição da lógica, que necessitaria de menos testes.

Após a recepção bem sucedida de um frame, é gerado um sinal para o módulo acoplado, indicando o evento. Neste caso, pode haver outro módulo diretamente ligado a ele, ou o Mailbox implementado neste trabalho. Os segmentos dos frames já estão separados, e são acessados de forma paralela. Em caso de frames no formato padrão, os 18 bits do identificador extendido são lixo. No data field, os dados estão alinhados a esquerda, e tem o comprimento de acordo com o definido no segmento length, em bytes.

Este Mailbox compara os dados do frame recebido com suas máscaras para verificar se essa mensagem é para este nodo, e para qual módulo de destino seria, encaminhando-a para uma porta de saída especificada e gerando um evento.

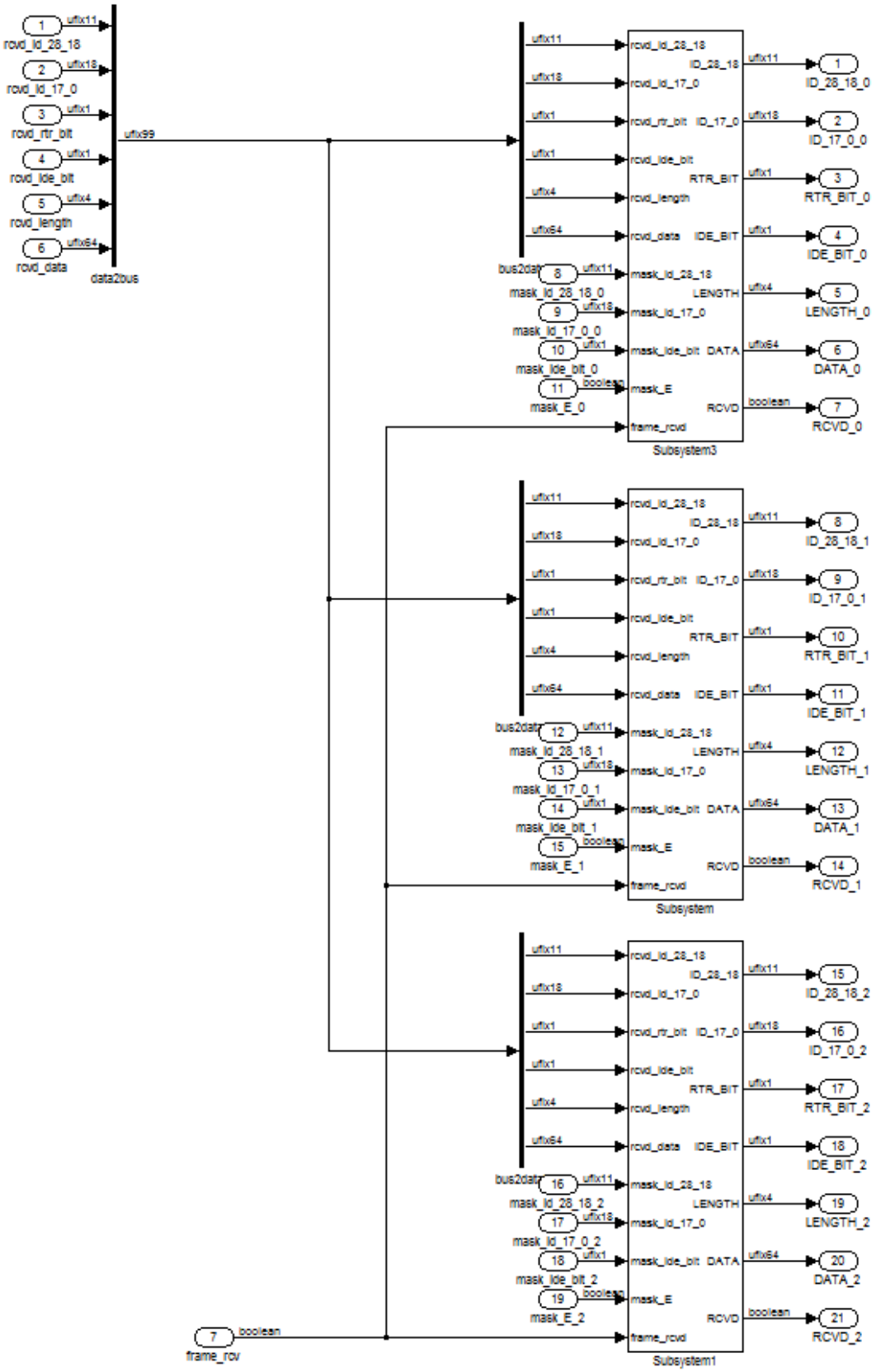

Figura 5.4: Mailbox com 3 saídas

5.4 Escalonador e buffer de transmissão

O buffer de transmissão é composto por um banco de registradores com os campos da mensagem, conectados a uma lógica que os ordena e insere nos respectivos lugares, de acordo com a descrição do protocolo.

Esta lógica repassa o frame para um shift register, que é deslocado pela lógica sempre que necessário. Neste registrador estão presentes os segmentos de SOF até data field, sem bit stuffing. O controle irá inserir o stuff bit em tempo de execução, e após transmitir até o último bit do data field, ou data length para remote frames, é utilizado o shift register presente no módulo de cálculo do CRC para transmitir o respectivo segmento.

O shift register sempre é setado imediatamente antes de começar a tentativa de transmissão de um frame.

Foi desenvolvido um escalonador, que tem função semelhante ao mailbox para recepção. Pode-se ter vários módulos acoplados a ele, com frames a transmitir, e ele escolherá um para repassar ao controlador. O critério utilizado foi de transmitir a mensagem de maior prioridade em espera, visando à transparência entre módulo acoplado e o barramento. Assim para o módulo, seria como se ele tivesse seu próprio controlador anexado ao barramento, já que se ele tivesse uma mensagem de menor prioridade, outra seria transmitida antes dela de qualquer modo. Quanto a mensagens de alta prioridade com uma taxa grande de transmissão, esse problema não é tratado, pois por parte da aplicação, é necessário um estudo sobre o sistema, para especificação de prioridades e quando elas devem ocorrer.

Após o começo da tentativa de transmissão de uma mensagem, o escalonador mantém fixa a mensagem a ser enviada até que ela perca a arbitração, seja completamente enviada ou ocorra um erro no barramento. Se durante esse intervalo de tempo, surgir uma mensagem de maior prioridade no módulo, está será a próxima a tentar ser transmitida, não importando o que tenha ocorrido na tentativa anterior.

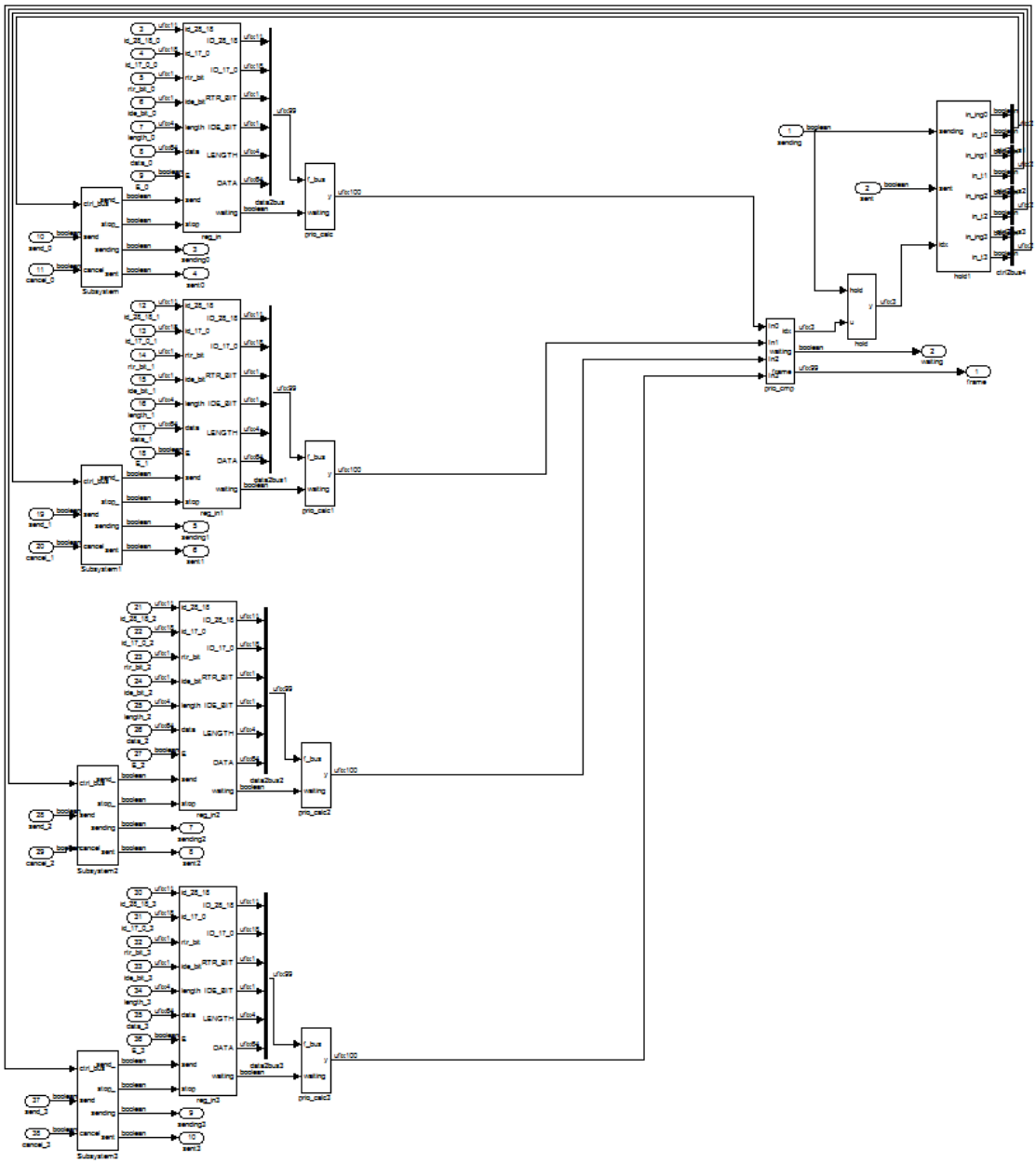

Figura 5.5: Escalonador com 4 entradas

5.5 Máquina de estado de controle

Apesar de todas os recursos oferecidos pelo Stateflow, foi decidido usar uma notação mais simples do que a definida por Harel, que apesar de ser muito rica, pode ser um tanto quanto complicada de visualizar certas vezes, e também por que a ferramenta de geração de código não dá suporte a todos os recursos.

Para implementação, foi utilizada a notação de Mealy, em que as ações são uma função das entradas e estados atuais, permitindo um maior conjunto de ações sem que se necessite de uma quantidade extra de estados, como seria na notação de Moore.

Após análise do protocolo, foi criado uma diagrama de transições que representa as interações entre os tipos de mensagem e estados do barramento. O diagrama pode ser visto na figura 5.6.

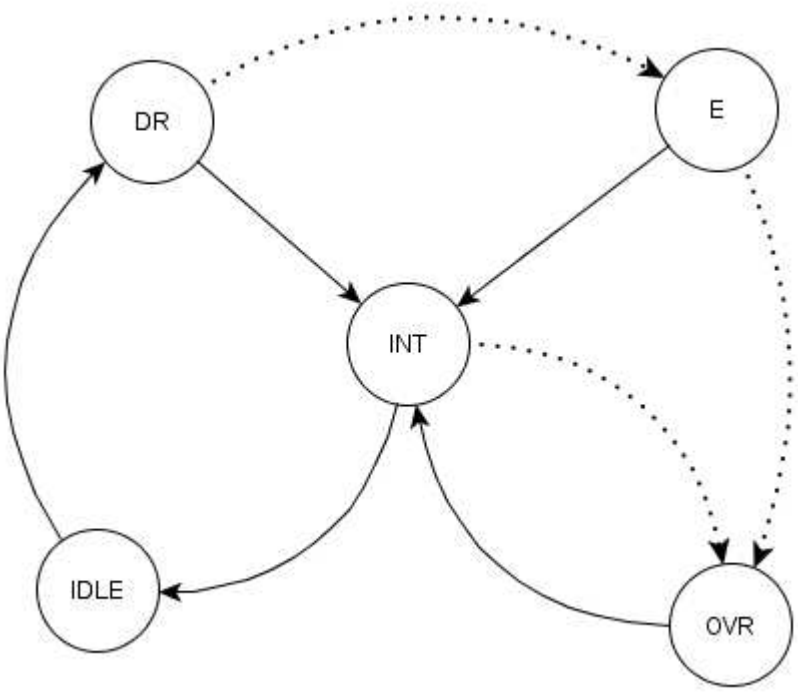

Figura 5.6: Diagrama de transições de estados do barramento

As transições com linhas constantes representam transições que ocorrem quando o estado chega ao fim de sua execução sem interrupções. As linhas pontilhadas representam transições que ocorrem antes do fim de sua execução, interrompendo-a. É causada por erros ou exceções presentes na especificação.

Internamente a cada estado, foi ele dividido em partes correspondentes ao seus segmentos. Um exemplo é o segmento do identificador extendido, na figura 5.7.

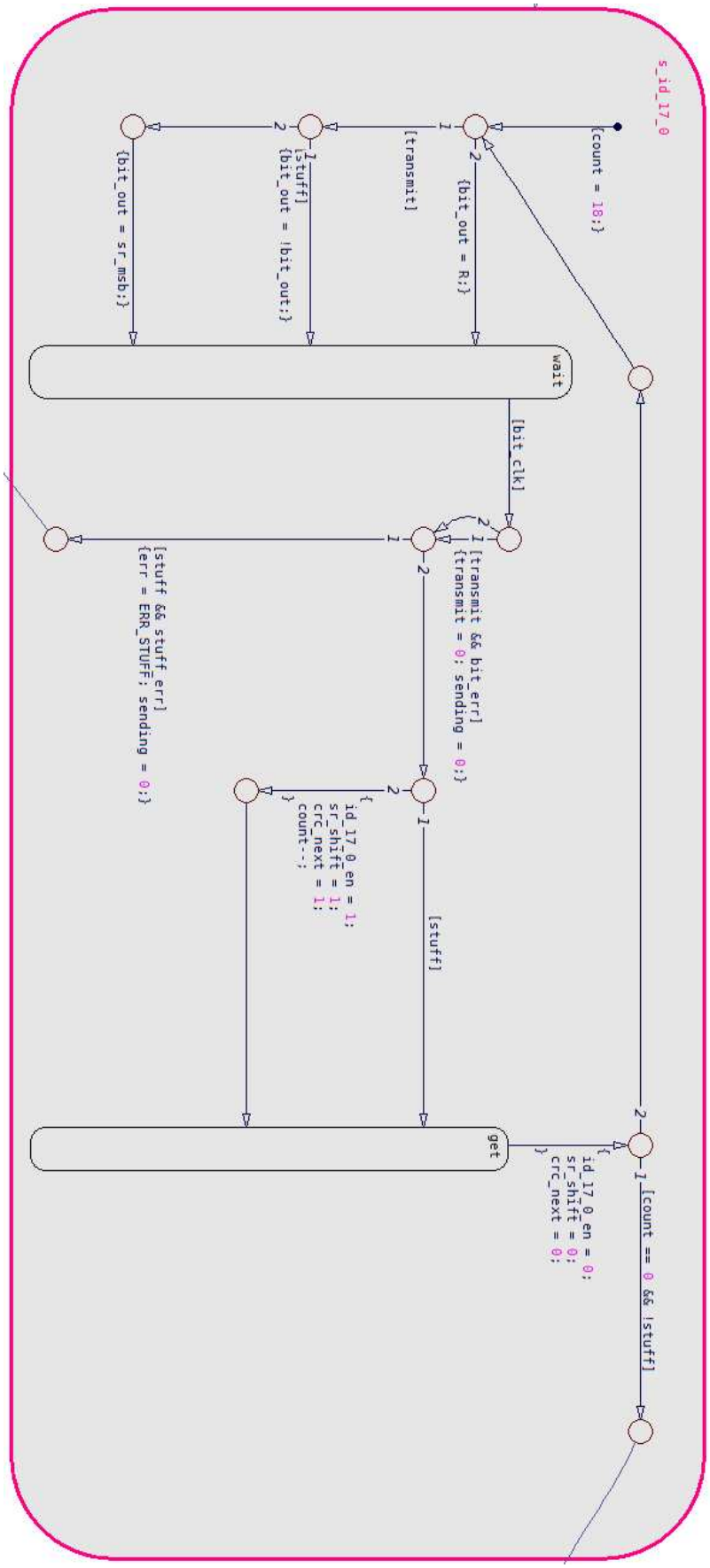

Figura 5.7: Estado para tratar o segmento do identificador extendido

E dentro de cada segmento há no geral dois estados. Um que espera pelo trigger gerado pelo módulo de bit timming, que representa o ponto de amostragem do bit, e outro que serve para auxiliar na geração de sinais de controle. Há a possibilidade, dependendo da característica do estado de haver loops dentro deles, para segmentos maiores que 1 bit. Como se pode ver também, devido às características da definição de Mealy, as ações ficam nas transições. Assim é possível no mesmo ciclo testar se houve erros no barramento e decidir as ações a serem tomadas. No ciclo seguinte as saídas já estão setadas, respeitando uma definição do protocolo de que é necessário no máximo dois unidades de tempo para o processamento das entradas.

Apesar de que basicamente todas as transações são feitas entre os estados mais baixos da hierarquia, foi se utilizado um modelo de hierarquia dos estados, recurso presente no Stateflow e definido por Harel, para fins organizacionais.

Também foi mantido um padrão semelhante ao acima entre todos os segmentos do protocolo, para facilitar o desenvolvimento e o manter mais entendível e manutenível.

A visão geral da máquina de estados segue na figura 5.8.

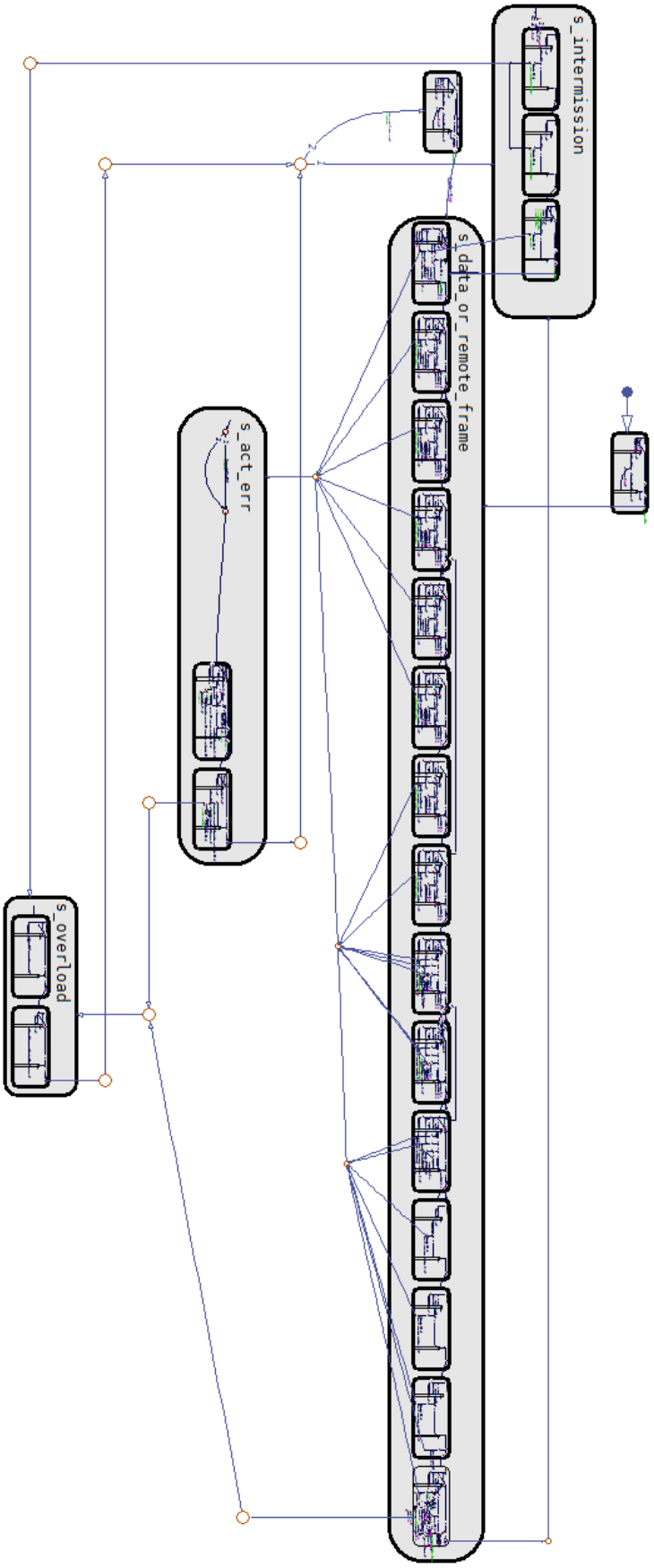

Figura 5.8: Máquina de estados de controle do módulo

6 RESULTADOS

6.1 Simulação

Foi feita uma simulação, no Simulink, para verificar as funcionalidades básicas do controlador, mailbox e escalonador.

Para tal, foi utilizado 3 nodos, 1 com apenas o controlador e os outros dois com os módulos auxiliares desenvolvidos neste trabalho. A rede se organiza como na figura 6.1.

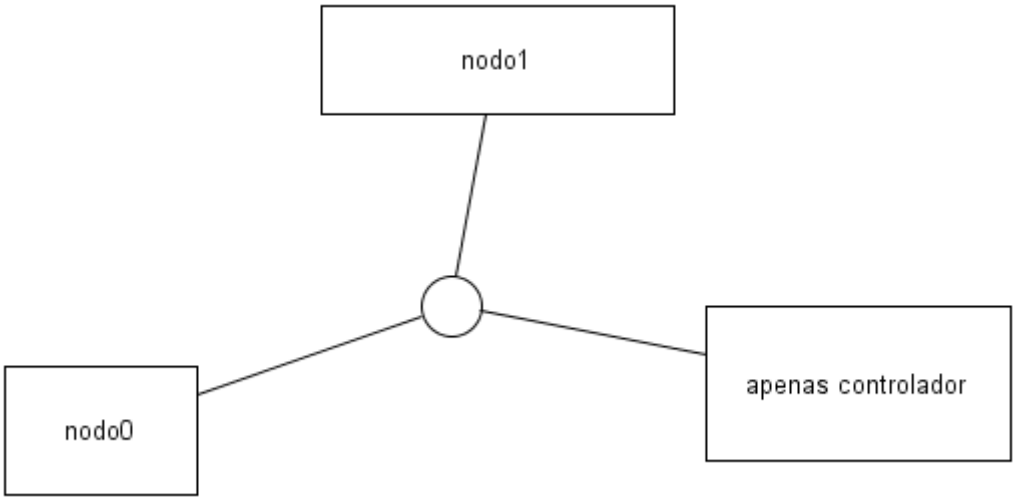

Figura 6.1: Rede usada na simulação

Para cada nodo foi configurados frames para transmissão e recepção.

Para o nodo0 as configurações são:

- Transmissão:
  - id_28_18 = 1700, id_17_0 = 17, extendido e length = 8;
  - id_28_18 = 256, padrão e length = 1;
  - id_28_18 = 1000, padrão e length = 3;
- Recepção:
  - id_28_18 = 1024, padrão;
  - id_28_18 = 512, id_17_0 = 215, extendido;
  - id_28_18 = 1700, padrão.

Para o nodo1 são:

- Transmissão:
  - id_28_18 = 1024, id_17_0 = 4201, extendido e length = 5;
  - id_28_18 = 1700, padrão e length = 7;
  - id_28_18 = 512, id_17_0 = 215, extendido e length = 2;
- Recepção:
  - id_28_18 = 256, padrão;
  - id_28_18 = 1000, padrão;
  - id_28_18 = 1536, id_17_0 = 6351, extendido;

Para o nodo sem os módulos auxiliares:

- Transmissão de:
  - id_28_18 = 1024, padrão e length = 4;
- E após o sucesso deste transmite:
  - id_28_18 = 1536, id_17_0 = 6351, extendido e length = 6;

Analisando o sistema, pode-se saber a ordem em que estas mensagens são enviadas, que é a seguinte:

1. id_28_18 = 256, padrão e length = 1 (nodo0);
2. id_28_18 = 512, id_17_0 = 215, extendido e length = 2 (nodo1);
3. id_28_18 = 1000, padrão e length = 3 (nodo0);
4. id_28_18 = 1024, padrão e length = 4 (nodo sem módulos);
5. id_28_18 = 1024, id_17_0 = 4201, extendido e length = 5 (nodo 1);
6. id_28_18 = 1536, id_17_0 = 6351, extendido e length = 6 (nodo sem módulos);
7. id_28_18 = 1700, padrão e length = 7 (nodo1);
8. id_28_18 = 1700, id_17_0 = 17, extendido e length = 8 (nodo0);

Com a execução da simulação obtemos os sinais no barramento e nas portas de transmissão dos módulos como mostra a figura 6.2.

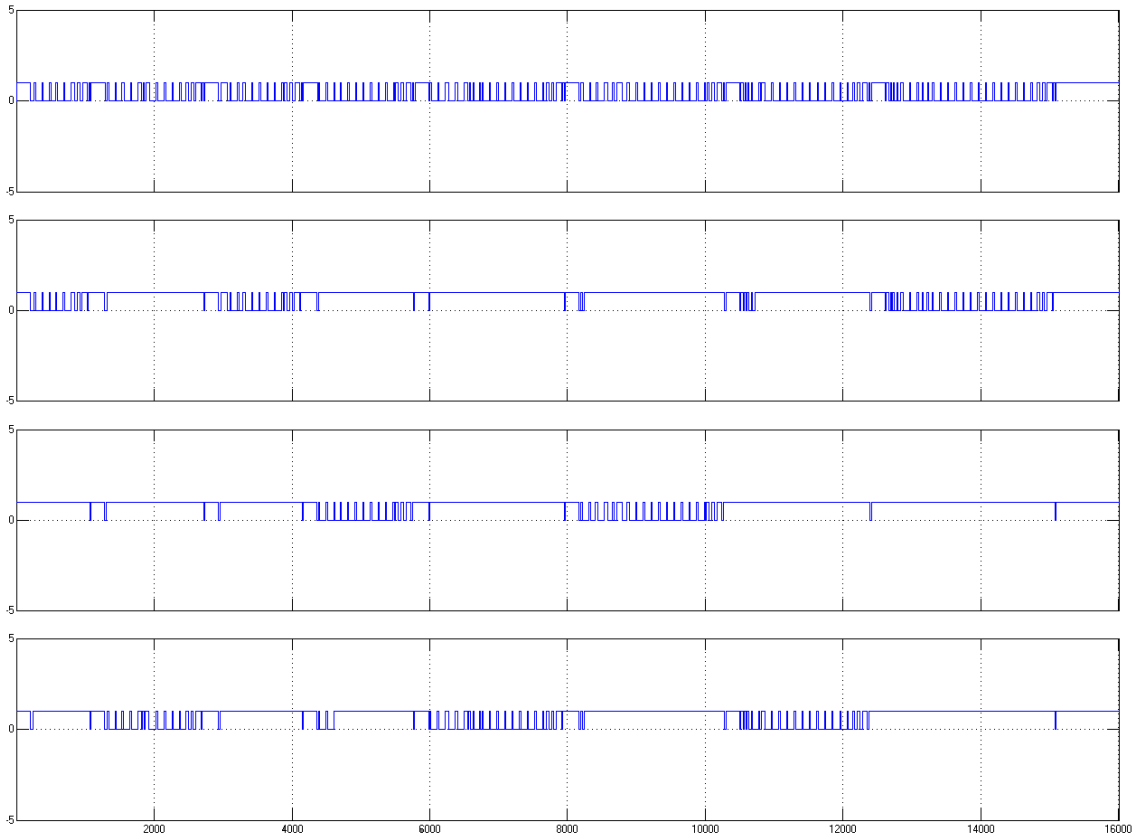

Figura 6.2: Resultado da simulação

A primeira linha mostra o valor no barramento CAN, que é a entrada nos módulos. As linhas seguinte representam respectivamente o nodo0, o nodo sem módulos auxiliares e o nodo1. Pode-se notar sinais de acknowledge, e tentativas de transmissão onde se perde na arbitração. Com este teste também é possível se notar que o escalonador fez corretamente sua tarefa, de transmitir antes mensagens de prioridade mais alta em espera primeiro.

Também foi adicionado pequenos delays aleatórios para simular desvios de fase no barramento.

A partir desse sistema gerou VHDL, assim como o testbench para o sistema, já que a ferramenta também possui geração de código para isso, assim como faz uma comparação entre os resultados esperados, que seriam os do Simulink, com os resultados gerados pela simulação comportamental do código. Na figura 6.3, pode-se ver a simulação comportamental do VHDL e as saídas do sistema, assim como nas ultimas linhas os valores esperados.

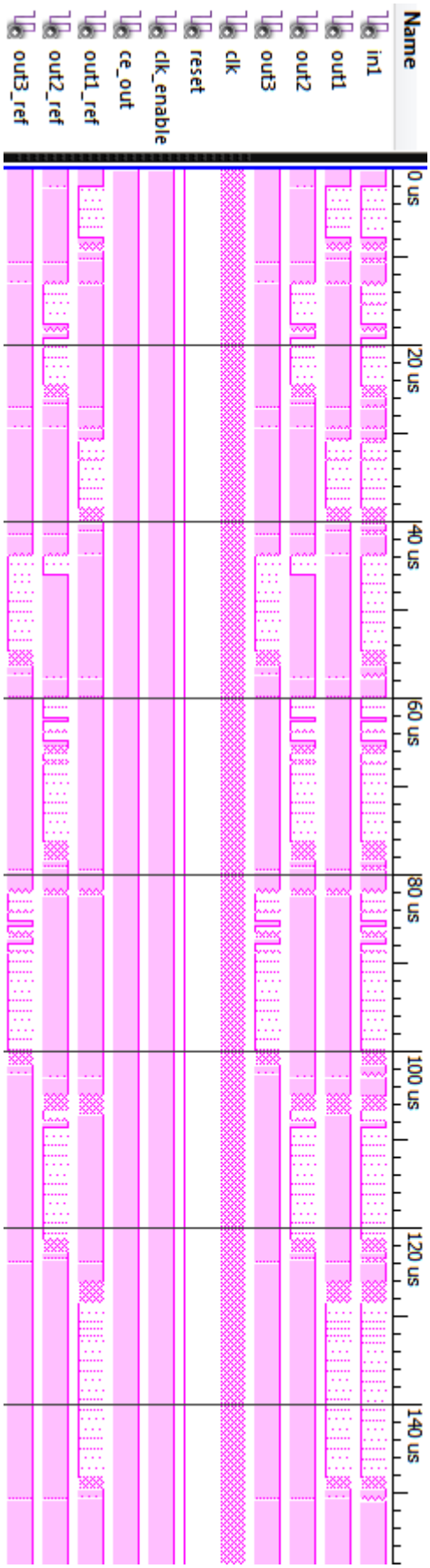

Figura 6.3: Simulação comportamental do VHDL gerado

Foram-se gerados também vetores de entrada, a partir de uma função do Matlab que montava frames CAN, feito numa etapa preliminar do desenvolvimento do projeto, e vetores retirados do testbench de um código Verilog do protocolo que será comentado nas próximas sub-seções, para verificar se os pacotes aceitos pelo protocolo estavam em conformidade, e que o funcionamento do módulo não fosse causado devido o teste ter sido feito com dois componentes de mesma origem. As simulações geraram resultados dentro do previsto. Pacotes corretos foram aceitos, inserção de erros foram detectadas e sinalizadas.

6.2 Geração do código e síntese

Para a geração do VHDL foi utilizado um dos nodos da simulação acima, removendo-se apenas os blocos utilizados para geração dos vetores de teste. Para a geração de código foi se usado um escalonador com 2 entradas e um mailbox com 3 buffers. A geração de código gerou em torno de 170 arquivos VHDL.

Para a síntese desta e da próxima seção será usado um Spartan 3E, device xc3s1600e, package fg484, para se ter uma plataforma fixa para comparação entre os resultados, e por também se ter licença sempre disponível para estas plataformas.

Os resultados da síntese se seguem nas figuras 6.4.

```
=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# FSMs                                          : 30
# Adders/Subtractors                            : 9
 4-bit adder                                    : 1
 7-bit subtractor                               : 1
 8-bit adder                                    : 4
 8-bit addsub                                   : 3
# Registers                                     : 841
 Flip-Flops                                     : 841
# Comparators                                   : 17
 11-bit comparator equal                        : 3
 18-bit comparator equal                        : 3
 31-bit comparator less                         : 1
 4-bit comparator greater                       : 2
 7-bit comparator greater                       : 1
 8-bit comparator equal                         : 1
 8-bit comparator greater                       : 5
 8-bit comparator less                          : 1
# Multiplexers                                  : 3
 8-bit 16-to-1 multiplexer                      : 3
# Xors                                          : 6
 1-bit xor2                                     : 6

=========================================================================
=========================================================================
Final Register Report

Macro Statistics
# Registers                                     : 891
 Flip-Flops                                     : 891

=========================================================================
```

```
Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 10.705ns (Maximum Frequency: 93.416MHz)
   Minimum input arrival time before clock: 11.592ns
   Maximum output required time after clock: 13.279ns
   Maximum combinational path delay: 14.167ns

=========================================================================
```

Figura 6.4: Resultado da síntese

Com a síntese deste módulo, conseguiu se atingir uma freqüência de operação apropriada para o funcionamento na velocidade máxima exigida pelo sistema. Por exemplo, utilizando-se 25 unidades de tempo de bit, e a velocidade máxima do barramento, 1Mbps, seria necessário uma freqüência de operação de 25 MHz. Este sistema supre facilmente está necessidade, precisando-se ainda utilizar um pre-scaler para ajustar a velocidade de funcionamento.

6.3 Comparação com outro VHDL

Para compará-la a qualidade do sistema gerado com uma ferramenta de desenvolvimento baseada a modelos, será utilizada um código VHDL encontrado no site OpenCores. Este código VHDL, que não foi testado para verificar se está funcional, é uma tradução do verilog feito por Igor Mohor, que havia sido validado e era funcional. Apesar disso considera-se válida a comparação para fins de verificar as diferenças de código gerado a mão e um gerado automaticamente. Esta implementação não esta bem documentada, então se necessitou inferir alguns recursos que ele provê. Aparentemente possui um mecanismo de FIFO, que no verilog era uma memória RAM, mas nesse não se conseguiu definir. Apesar destas diferenças estruturais, a grandeza a ser verificada neste caso é a freqüência, para se verificar se há uma grande diferença quanto à otimização temporal de um código gerado manualmente com um gerado automaticamente. Alguns resultados da síntese se seguem.

```
=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# ROMs                                          : 1
 8x3-bit ROM                                    : 1
# Adders/Subtractors                            : 18
 2-bit adder                                    : 1
 3-bit adder                                    : 3
 3-bit subtractor                               : 2
 4-bit adder                                    : 4
 4-bit subtractor                               : 1
 6-bit subtractor                               : 2
 7-bit adder                                    : 1
 7-bit subtractor                               : 1
 8-bit subtractor                               : 1
 9-bit addsub                                   : 2
# Counters                                      : 19
 2-bit up counter                               : 1
 3-bit up counter                               : 10
 4-bit up counter                               : 2
 5-bit up counter                               : 2
 6-bit up counter                               : 2
 7-bit up counter                               : 1
 7-bit updown counter                           : 1
# Registers                                     : 517
 Flip-Flops                                     : 517
# Comparators                                   : 44
 15-bit comparator not equal                    : 1
 2-bit comparator less                          : 1
 3-bit comparator equal                         : 2
 3-bit comparator less                          : 8
 4-bit comparator equal                         : 1
 4-bit comparator less                          : 4
 5-bit comparator equal                         : 1
 5-bit comparator greater                       : 1
 6-bit comparator equal                         : 2
 6-bit comparator greater                       : 2
 6-bit comparator less                          : 1
 7-bit comparator equal                         : 1
 8-bit comparator equal                         : 1
 8-bit comparator greatequal                    : 3
 8-bit comparator less                          : 1
 8-bit comparator lessequal                     : 2
 9-bit comparator greatequal                    : 7
 9-bit comparator greater                       : 3
 9-bit comparator less                          : 2
# Multiplexers                                  : 16
 1-bit 16-to-1 multiplexer                      : 2
 1-bit 19-to-1 multiplexer                      : 2
 1-bit 39-to-1 multiplexer                      : 1
 1-bit 64-to-1 multiplexer                      : 3
 1-bit 8-to-1 multiplexer                       : 8
# Xors                                          : 101
 1-bit xor2                                     : 101

=========================================================================
```

Figura 6.5: Resultados da síntese do VHDL manual

```
=========================================================================
Final Register Report

Macro Statistics
# Registers                                          : 541
 Flip-Flops                                          : 541

=========================================================================
Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 10.274ns (Maximum Frequency: 97.334MHz)
   Minimum input arrival time before clock: 9.609ns
   Maximum output required time after clock: 6.380ns
   Maximum combinational path delay: 6.665ns
```

Figura 6.6: Resultados da síntese do VHDL manual (cont.)

Pode-se notar que a freqüência máxima obtida em ambos não se difere muito, mostrando que o modelo pode atingir desempenhos próximos a de códigos gerados manualmente, e com uma facilidade maior de desenvolvimento e compreensão.

7 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho foi apresentado à modelagem do protocolo CAN, um protocolo de comunicação serial altamente usado em meios industriais, utilizando-se uma ferramenta de desenvolvimento baseada em modelos, o Simulink, com o apoio do Stateflow para geração de máquinas de estados complexas, para geração automática de VHDL, para que possa ser utilizado em ASICs ou FPGAs.

Apesar de estas ferramentas ajudarem no desenvolvimento de sistemas, digitais ou não, poupando a etapa de codificação, muito suscetível a erros e de mais difícil manutenção e diagnosticação de erros, não diminui a complexidade no planejamento e elaboração do trabalho. Todo projeto ainda deve ser modelado previamente com a interação entre os blocos e suas funcionalidades, mas já é possível trabalhar o desenvolvimento em cima deste modelo. A modelagem do controle, usando-se máquinas de estados também é facilitada utilizando-se estas ferramentas, sendo possível simulá-lo e analisar o seu comportamento de maneira interativa e de fácil correção de erros.

No entanto, vale ressaltar que as ferramentas utilizadas não possuem como objetivo principal o desenvolvimento de sistemas digitais, e sim simulação de sistemas complexos, que muitas vezes possuem características e recursos, não compatíveis com o desenvolvimento de circuitos digitais. Como é uma ferramenta muito genérica, com freqüência ocorria erros de configuração em módulos ou ferramentas, que se não fossem vistos cedo trariam muitas dificuldades de correção. A interface gráfica também por vezes não é das melhores para o desenvolvimento de circuitos digitais. A interação entre os blocos gera um número excessivo de segmentos, o que em modelos mais complexos gera uma poluição visual grande. E havia ainda casos de bugs.

Outro ponto que vale ressaltar, foi que durante o trabalho se conheceu uma maneira diferente de se modelar máquinas de estados, diferente das tradicionais e aprendidas durante a graduação (Mealy e Moore), a de Harel. A ferramenta Stateflow se baseia neste método, apesar de ainda não conseguir dar todo suporte a ela, e menos ainda o HDL Coder, parece uma solução bem interessante a ser estudada sua viabilidade na utilização de sistemas digitais.

Sobre a implementação em si, ainda há pontos a melhorar. Na atual configuração, há um uso talvez desnecessário de registradores, já que sempre se tem reservados para a extensão do identificador e 64 de dados, que nem sempre serão utilizados. Poderia ser feito algum método, para gerenciar os buffers e diminuir o desperdício. Atualmente também, a comunicação entre o controlador e outros módulos é paralela, utilizando um grande número de linha para a transferência de dados.

Este foi o primeiro projeto de um sistema digital pouco mais complexo feito por mim, e já havia o desejo de fazer um trabalho nesta área, pois não houve a oportunidade durante cadeiras feitas e em lugares onde trabalhei. Também foi interessante desenvolver um projeto utilizando ferramentas baseadas em modelos, que se mostrou um método bem eficiente, apesar de não se ter utilizado uma ferramenta com este propósito.

REFERÊNCIAS

Reynari, L.M., Cucinotta, F., Serra, A. e Lavagno, L. (2001) “A Hardware/Software Co-design Flow and IP Library Based of Simulink”. In: Design Automation Conference, 2001. Proceedings, pág. 593-598.

Weiss, K. A., E. C. Ong and N. G. Leveson, 2003, "Reusable Specification Components for Model-Driven Development". In International Conference on System Engineering (INCOSE '03), Crystal City, Virginia USA.

Douglas C. Schmidt, " Model-Driven Engineering," IEEE Computer, pp. 25-31, February, 2006

Bran Selic. 2003. The Pragmatics of Model-Driven Development. IEEE Software. Volume 20, Issue 5 (September 2003), 19-25.

Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. 2004. Reuse of software in distributed embedded automotive systems. In Proceedings of the 4th ACM international conference on Embedded software (EMSOFT '04). ACM, New York, NY, USA, 203-210.

Robert Siegmund, Dietmar Müller, "A Novel Synthesis Technique for Communication Controller Hardware from declarative Data Communication Protocol Specifications," Design Automation Conference, p. 602, 39th Design Automation Conference (DAC'02), 2002

Van Beeck, K. and Heylen, F. and Meel, J. and Goedemé, T., “Comparative study of Model-based hardware design tools”, 2010

Vizhanyo, A. and Neema, E. and Kalmar, Z. and Shi, F. and Karsai, G., “Model-Driven Software Development of Model-Driven Tools: A Visually-Specified Code Generator for Simulink/Stateflow”, 2008

MathWorks, Stateflow® and Stateflow® Coder™ 7 User’s Guide

MathWorks, Simulink® 7 User’s Guide

MathWorks, Simulink® HDL Coder™ 2 User’s Guide

Carvalho, F.C., Jansch-Porto, I., Freitas, E.P., Pereira, C.E., “The TinyCAN: an optimized CAN controller IP for FPGA-based platforms”, Proceedings of 10th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2005, September 19-22, 2006, Catania, Italy

Harel, D., Statecharts: A visual formalism for complex systems, Science of computer programming, 1987, 231-274

APÊNDICE GERAÇÃO DE CÓDIGO A PARTIR DE UM MODELO SIMPLES DO SIMULINK

Neste apêndice será mostrado como se gera código VHDL a partir do Simulink com as ferramentas utilizadas neste trabalho.

Na figura a seguir há um modelo simples do Simulink, que possui uma máquina de estados, um circuito combinacional (somador) e um delay de um ciclo de sinal (registrador).

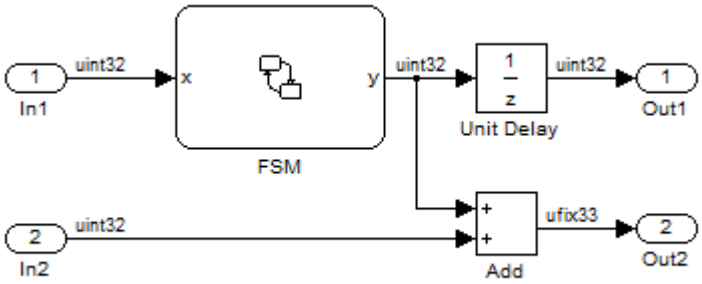

Figura: Modelo do Simulink

E na figura a seguir, a máquina de estados modelada:

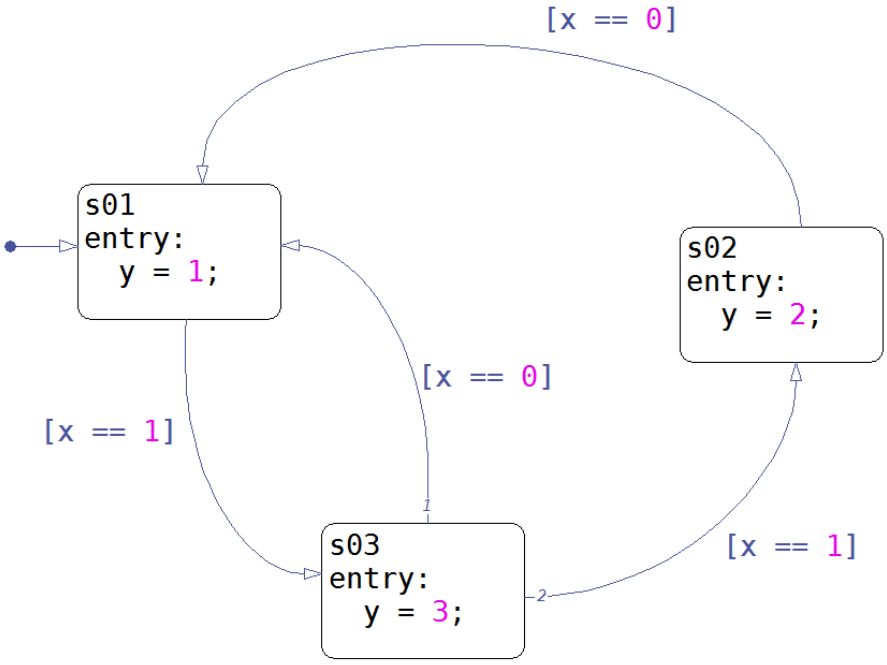

Figura: Máquina de estados do Stateflow

O HDL Coder possui uma ferramenta que auxilia na configuração do modelo para geração de código, assim como opções de como este código deve ser gerado. Está ferramenta se chama HDL Workflow Advisor, que com ela também é possível integrar com outras ferramentas de design de hardware digital, como por exemplo, o ISE da Xilinx.

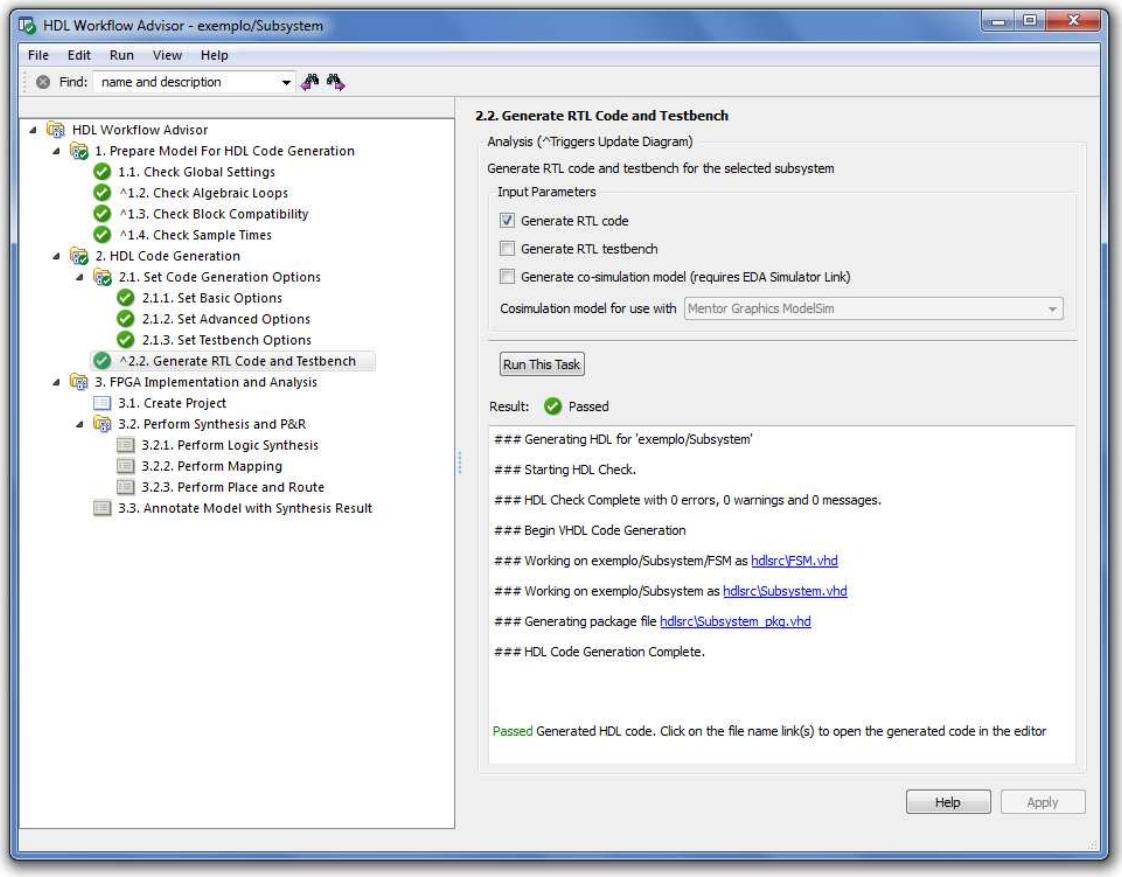

Figura: HDL Workflow Advisor

A seguir, o código gerado para o modelo exemplificado.

Máquina de estados:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE work.Subsystem_pkg.ALL;

ENTITY FSM IS
  PORT( clk                               :   IN    std_logic;
        reset                             :   IN    std_logic;
        enb                               :   IN    std_logic;
        x                                 :   IN
std_logic_vector(31 DOWNTO 0);  -- uint32
        y                                 :   OUT
std_logic_vector(31 DOWNTO 0)  -- uint32
        );
END FSM;
```

```
ARCHITECTURE rtl OF FSM IS

  -- Signals
  SIGNAL x_unsigned                       : unsigned(31 DOWNTO
0);  -- uint32
  SIGNAL is_FSM                           : T_state_type_is_FSM;
-- uint8
  SIGNAL y_tmp                            : unsigned(31 DOWNTO
0);  -- uint32
  SIGNAL y_reg                            : unsigned(31 DOWNTO
0);  -- uint32
  SIGNAL is_FSM_next                      : T_state_type_is_FSM;
-- enumerated type (3 enums)
  SIGNAL y_reg_next                       : unsigned(31 DOWNTO
0);  -- uint32

BEGIN
  x_unsigned <= unsigned(x);

  FSM_1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      is_FSM <= IN_s01;
      y_reg <= to_unsigned(1, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        is_FSM <= is_FSM_next;
        y_reg <= y_reg_next;
      END IF;
    END IF;
  END PROCESS FSM_1_process;

  FSM_1_output : PROCESS (is_FSM, x_unsigned, y_reg)
  BEGIN
    is_FSM_next <= is_FSM;
    y_reg_next <= y_reg;

    CASE is_FSM IS
      WHEN IN_s01 =>
        IF x_unsigned = 1 THEN
          is_FSM_next <= IN_s03;
          y_reg_next <= to_unsigned(3, 32);
        END IF;
      WHEN IN_s02 =>
        IF x_unsigned = 0 THEN
          is_FSM_next <= IN_s01;
          y_reg_next <= to_unsigned(1, 32);
        END IF;
      WHEN IN_s03 =>
        IF x_unsigned = 0 THEN
          is_FSM_next <= IN_s01;
          y_reg_next <= to_unsigned(1, 32);
        ELSIF x_unsigned = 1 THEN
          is_FSM_next <= IN_s02;
          y_reg_next <= to_unsigned(2, 32);
```

```
        END IF;
      WHEN OTHERS =>
        is_FSM_next <= IN_s01;
        y_reg_next <= to_unsigned(1, 32);
    END CASE;

  END PROCESS FSM_1_output;

  y_tmp <= y_reg_next;

  y <= std_logic_vector(y_tmp);

END rtl;
```

Modelo:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY Subsystem IS
  PORT( clk                               :   IN    std_logic;
        reset                             :   IN    std_logic;
        clk_enable                        :   IN    std_logic;
        In1                               :   IN
std_logic_vector(31 DOWNTO 0);  -- uint32
        In2                               :   IN
std_logic_vector(31 DOWNTO 0);  -- uint32
        ce_out                            :   OUT   std_logic;
        Out1                              :   OUT
std_logic_vector(31 DOWNTO 0);  -- uint32
        Out2                              :   OUT
std_logic_vector(32 DOWNTO 0)  -- ufix33
        );
END Subsystem;


ARCHITECTURE rtl OF Subsystem IS

  -- Component Declarations
  COMPONENT FSM
    PORT( clk                             :   IN    std_logic;
          reset                           :   IN    std_logic;
          enb                             :   IN    std_logic;
          x                               :   IN
std_logic_vector(31 DOWNTO 0);  -- uint32
          y                               :   OUT
std_logic_vector(31 DOWNTO 0)  -- uint32
          );
  END COMPONENT;

  -- Component Configuration Statements
  FOR ALL : FSM
    USE ENTITY work.FSM(rtl);
```

```
  -- Signals
  SIGNAL enb                              : std_logic;
  SIGNAL FSM_out1                         : std_logic_vector(31
DOWNTO 0);  -- ufix32
  SIGNAL FSM_out1_unsigned                : unsigned(31 DOWNTO
0);  -- uint32
  SIGNAL Unit_Delay_out1                  : unsigned(31 DOWNTO
0);  -- uint32
  SIGNAL In2_unsigned                     : unsigned(31 DOWNTO
0);  -- uint32
  SIGNAL Add_out1                         : unsigned(32 DOWNTO
0);  -- ufix33

BEGIN
  u_FSM : FSM
    PORT MAP( clk => clk,
              reset => reset,
              enb => clk_enable,
              x => In1,  -- uint32
              y => FSM_out1  -- uint32
              );

  FSM_out1_unsigned <= unsigned(FSM_out1);

  enb <= clk_enable;

  Unit_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay_out1 <= to_unsigned(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        Unit_Delay_out1 <= FSM_out1_unsigned;
      END IF;
    END IF;
  END PROCESS Unit_Delay_process;


  Out1 <= std_logic_vector(Unit_Delay_out1);

  In2_unsigned <= unsigned(In2);

  Add_out1 <= resize(FSM_out1_unsigned, 33) +
resize(In2_unsigned, 33);

  Out2 <= std_logic_vector(Add_out1);

  ce_out <= clk_enable;

END rtl;
```

ANEXO A TRABALHO DE GRADUAÇÃO 1

Implementação do Protocolo CAN utilizando Simulink para geração automática de VHDL

Matheus Vogel Pinto, Luigi Carro

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

{mvpinto,carro}@inf.ufrgs.br

Resumo. *Este trabalho apresentará a implementação do protocolo CAN, altamente usado em veículos e equipamentos médicos, em um alto nível de abstração utilizando MATLAB e Simulink, para geração automática de código em VHDL. Além de exigir um período menor de desenvolvimento e menos suscetividade a erros, com ferramentas baseada em modelos, é possível gerar códigos em diferentes linguagens, que possuiriam o mesmo comportamento e utilizando o mesmo modelo. Conseguindo gerar um HDL para esse protocolo, seria possível em apenas um ASIC ou FPGA, ter toda a aplicação e o controlador do protocolo em um componente, sem a necessidade de componentes extras, como um só para aplicação e outro só para o controlador. Será ainda implementado mais um módulo que permite a conexão de vários módulos ao mesmo controlador de comunicação.*

Abstract. *This work will present an implementation of the CAN protocol, highly used in vehicles and medical equipments, in a high level of abstraction using MATLAB and Simulink, to generate automatically VHDL code. Beyond demands a shorter development period and less susceptible to errors, with model-based tools is possible to generate code to different languages, which has the same behavior and using the same model. Generating a HDL code to this protocol, would be possible have in an ASIC or FPGA, the whole application and the protocol controller in one single device, without the need for extra devices, like one for the application and other to the controller. Will be implemented also one more module, that handles the connection of the controller with several application modules.*

1. Introdução

Já se passou o tempo em sistemas embarcados eram apenas microcontroladores simples com pouca RAM ou hardwares pequenos, e passaram a ser componentes ou sistemas que executam tarefas complexas e cada vez mais demandam novas aplicações e usos, como DVD Players, smartphones, sistemas de controles automotivos. E esses sistemas muitas vezes são compostos de várias unidades de processamento, que devem cooperar para atingir um objetivo, e ainda, em certos casos, suprir os requisitos de tempo real. Por exemplo, em carros comuns é possível encontrar em torno de 50 dispositivos que controlam diferentes funções, desde o controle de freios e injeção eletrônica até equipamentos de mídia e entretenimento.

Devido à evolução tecnológica destes dispositivos e a demanda por novas funcionalidades, desenvolvedores devem se adaptar cada vez mais freqüentemente a novas plataformas de trabalho e desenvolvimento de novas aplicações. Este tempo reduzido e o aumento da complexidade, causam um aumento na chance de se cometerem erros, comprometendo o tempo de desenvolvimento e a qualidade do produto final. Devido a isso, é exigido métodos eficientes de desenvolvimento, como por exemplo, automação de tarefas e reuso de bloco já previamente desenvolvido e validados.

Sistemas embarcados, ou de controle, modernos são complexos, e para domínios específicos de aplicação, geralmente possuem sub-sistemas e interações semelhantes. As similaridades entre esses sistemas torna a criação de um sistema genérico de desenvolvimento guiado a modelos muito atrativo, permitindo uma fácil adaptação do sistema a uma nova plataforma, ou a criação de um novo utilizando componentes já desenvolvidos. Com as adaptações, esses modelos podem ser executados e verificados em ambientes de simulação, e transformado em software/hardware manual ou automaticamente.

O reuso de componentes em hw/sw já é uma solução parcial para os custos e tempos no desenvolvimento de sistemas complexos. Mas isto já causou várias perdas, como por exemplo a NASA, que perdeu bilhões devido ao reuso de software com adaptações mal-sucedidas. Esses casos mal-sucedidos talvez se devam ao fato do nível em que essa técnica é utilizada. Geralmente esse reuso em nível de código é bem problemático, mas que poderia ser mais efetivo e seguro em fases anteriores de desenvolvimento num ambiente de desenvolvimento guiado a modelos.

Com um desenvolvimento guiado a modelos, a especificação é estruturada como uma hierarquia de modelos e sua interação entre estes. Isto facilita a definição de requisitos e restrições em nível de sistemas ate os detalhes de design e implementação, auxilia na garantia de propriedades de sistemas e reduz custos de mudança de implementação e revalidação do sistema.

Outra vantagem de uma especificação guiada a modelos é que modelos são expressos com conceitos menos limitantes à tecnologia alvo e mais próximos ao domínio dos problemas do que as linguagens de programação mais populares. Isto faz os modelos serem mais fáceis de serem especificados, entendidos e mantidos, e em certos casos podem permitir que a produção do sistema fique a cargo de um expert da area do problema, ao invés de alguém com conhecimento sobre as tecnologias. E também são menos sensíveis a tecnologia alvo, permite fácil portabilidade para novas plataformas ou versões.

Modelagem de software e geração automática de código já foram tentadas antes, mas com sucesso apenas em áreas específicas. Mas agora temos um melhor entendimento na modelagem de sistemas, esta técnica de desenvolvimento se torna mais útil que no passado, devido a que as ferramentas de automatização evoluíram, assim como surgiram padrões no ambiente industrial. Assim a linguagem do modelo toma o papel da implementação das linguagens de programação, assim como elas fizeram com as linguagens assembly.

Neste contexto, a MathWorks possui o Simulink, uma ferramenta de simulação e desenvolvimento guiado a modelos para sistemas embarcados e dinâmicos. Possui um

ambiente gráfico e uma série de blocos parametrizáveis para ajudar no desenvolvimento do sistema, assim como add-ons, que possuem já soluções parciais para certos domínios ou que permitam a implementação para plataformas alvo e geração automática de código. Alguns desses add-ons para geração automática de código, são por exemplo o Real Time Workshop, que gera código C para sistemas de tempo-real ou não, e o HDL Coder que gera código VHDL.

Então será proposto nesse trabalho a implementação de um protocolo de comunicação, mostrando que a partir de um modelo, pode-se gerar automaticamente código, tão confiáveis e quase tão eficiente quanto gerados manualmente.

2. Revisão Bibliográfica

Primeiramente será feita uma breve descrição de ferramentas disponíveis para implementação do projeto, e algumas de suas características chave. Em seguida será comentado alguns detalhes do protocolo CAN.

2.1. Ferramentas

O Simulink é uma ferramenta de simulação e desenvolvimento guiado a modelos para sistemas dinâmicos e embarcados, desenvolvida pela Mathworks. Prove um ambiente gráfico e interativo para desenvolvimento de sistemas, e ainda uma série de bibliotecas com blocos customizáveis, de diversas áreas como por exemplo controle, processamento de sinais, comunicações, etc.

Uma de suas bibliotecas é o Stateflow, que permite a criação de máquinas de estado, tabelas verdade e diagramas de fluxo. Este módulo permite descrever lógicas complexas de maneira natural e facilmente entendível.

Com essas duas ferramentas juntas formam um ambiente de desenvolvimento guiado a modelos para embarcados muito poderosa, pois permite a modelagem de sistemas complexos rapidamente, com menor suscetividade e já prove um ambiente de simulação.

A Mathworks ainda prove módulos para geração automática de código a partir de modelos do Simulink, para diversas linguagens alvo, e em certos casos, já otimizado para algumas arquiteturas. Duas delas são o HDL Coder e o Real-Time Workshop.

O HDL Coder gera código Verilog ou VHDL a partir de modelos do Simulink, códigos do Matlab e diagramas do Stateflow, que são bit-true e cicle-accurate. Alguns cuidados devem ser tomados, já que nem todos os blocos são sintetizáveis, e algumas técnicas devem ser tomadas com cautelas, como portas de trigger e enable, ou outras que acabam por tirar a propriedade de cicle-accurate e bit-true da simulação com o código sintetizado.

O Real-Time Workshop é uma ferramenta para geração e execução de código C para desenvolvimento e teste por algoritmos gerados pelo Simulink e Embedded Matlab code. Esse código gerado pode ser usado em sistemas de tempo real ou não, incluindo aceleração na simulação, rápida prototipação e testes hardware-in-the loop, e pode ser usado em qualquer microprocessador ou sistema operacional de tempo real.

Existem ainda outras que utilizam o Simulink em seu processo de automatização, seja por sua interface gráfica e conjunto de bibliotecas, seja por seu ambiente de simulação.

A Xilinx criou uma biblioteca para ser usada no Simulink, com uma série de blocos básicos às funções de DSP complexas, que foram desenvolvidos para gerarem lógicas rápidas e que utilizam pouca área. Esses blocos são de propriedade intelectual, então não se há acesso a suas implementações, e esse sistema deve ser composto exclusivamente pelos módulos destas bibliotecas. Esta ferramenta usa o ambiente gráfico do Simulink, e garante que o modelo sintetizado é bit-true e cicle-accurate com a simulação.

2.2. Protocolo CAN (Control Area Network)

O CAN é um protocolo de comunicação serial que suporta eficientemente controle de tempo real com um alto nível de segurança. Em eletrônicos automotivos são utilizadas velocidades de até 1 Mb/s.

Esse é um protocolo CSMA/CA (Carrier sense multiple access with collision avoidance). A mesma mídia de transmissão é utilizada por vários dispositivos, e os transmissores devem ouvir o barramento, para verificar se os dados recebidos são realmente os que estão sendo enviados. Caso seja diferente ele interrompe a transmissão e espera até a mídia estar livre para começar uma nova transmissão.

Para atingir transparência no design e flexibilidade de implementação, o CAN foi subdividido em diferentes camadas, de acordo com o modelo de referência ISO/OSI: Data Link Layer, dividido em Logical Link Control (LLC) e Medium Access Control (MAC) e Phisical Layer. A camada MAC é responsável controle de frames, realiza a arbitragem, verificação de erros, sinalização de erros e confinamento de falhas. A camada LLC prove serviços para transferência de dados e requisição de dados remotos, decidir quais pacotes recebidos são realmente para serem aceitos e prover gerência de recuperação e notificações de overload.

É definido que os níveis dos bit são denominados “recessivo” e “dominante”. Deve-se ao fato que quando há dois transmissores, quando um bit dominante é escrito no barramento, ele predominará sobre o recessivo, assim é possivel que um dos transmissores detecte esse ocorrido e pare a transmissão, esperando por um novo período de idle.

Existe 4 tipos de frames: DATA FRAME, REMOTE FRAME, ERROR FRAME e OVERLOAD FRAME.

DATA FRAME serve para se enviar dados pela rede, e o REMOTE FRAME para requisitar um determinado dado. Esses pacotes possuem identificadores, que também representam a prioridade da mensagem e são únicos, e são utilizados para a arbitração de quem prevalece na transmissão quando dois ou mais dispositivos começam a transmitir no começo de um período de idle. Possuem dispositivos para verificação de erros, como CRC e bit stuffing.

ERROR FRAME é enviado para rede quando os receptores encontram algum erro na transmissão. Podem ser ativos, que transmitem um pacote de erro assim que o erro é encontrado, ou passivo, que enviam erros em períodos específicos. Os erros

podem ser BIT ERROR, STUFF ERROR, CRC ERROR, FORM ERROR e ACKNOWLEDGEMENT ERROR. O protocolo tem mecanismos de confinamento de erros. Esse frame pode interromper um DATA ou REMOTE FRAME, logo que o erro é detectado, e os outros módulos reconhecem este frame pelo fato dele destruir os mecanismos de detecção de erros dos frames previamente citados.

OVERLOAD FRAME que é enviado quando algum dos módulos do sistema necessita de mais tempo para processar suas atividades e não tem como receber pacotes no momento. Há um limite para essas requisições, mas esses tipos de pacotes podem ser enviados em casos de certos acontecimentos na rede.

Na camada física, é necessário fazer a codificação e decodificação do bit, além do controle de tempo de cada um destes, que define a velocidade da rede e deve ser a mesma para todos dispositivos nela. Também a sincronização de bit, já que não há um clock global na rede, e possui tipos diferentes, dependendo das condições.

Um bit time é composto por diferentes segmentos: segmento de sincronização, o segmento de propagação, e dois segmentos de fase de bufferização, que entre eles é onde ocorre a amostragem do bit, e pode ter seus tamanhos modificados de acordos com as técnicas de sincronização.

Implementações deste protocolo em quaisquer tipos de HDL, foi apenas encontrada em Verilog, e uma versão "VHDLizada" deste. A Xilinx disponibiliza também uma versão em VHDL do protocolo, mas a licença para este módulo tem um custo elevado, em torno de 20.000 €.

3. Planejamento e implementação

Sobre este protocolo CAN, às vezes é utilizado um dispositivo de chamado de "mailbox", que permite que vários módulos dentro de um dispositivo utilizem o mesmo controlador de comunicação. Este módulo roteia as mensagens recebida para o módulo de destino, decide qual mensagem será enviada em seguida entre todas que estão em espera, e já responde automaticamente quando é requisitado por um REMOTE FRAME uma mensagem que já está em espera. Este módulo diminui a necessidade de processamento dos outros módulos no mesmo dispositivo e evita a repetição de pedaços iguais no mesmo.

Apesar de ser um módulo razoavelmente utilizado, e existirem controladores de CAN essa função, não foi encontrado VHDL pronto. A implementação desse sistema então permite que seja necessário apenas um driver que conecte o controlador com um barramento físico, sendo possível utilizar o mesmo módulo para tipos diferentes de mídia e ainda fazer modificações quando necessário. Este conjunto de módulos também é conhecido como FullCAN.

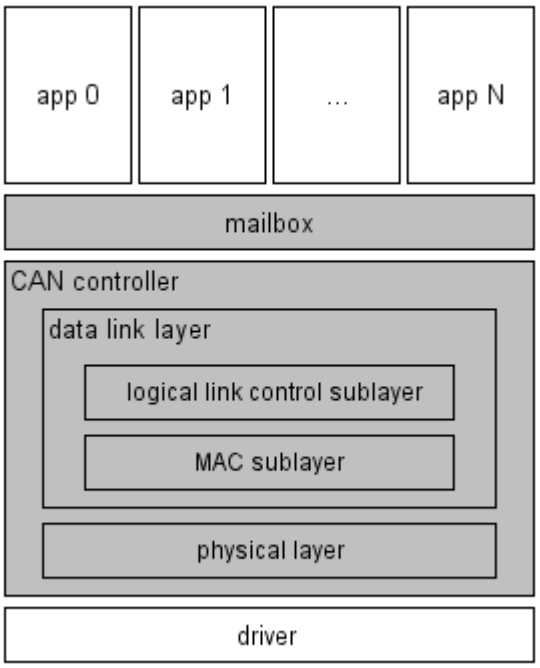

Figura 1. Exemplo de hierarquia de um sistema com este controlador

Neste trabalho será implementado o Protocolo CAN 2.0B, que dá a opção de identificadores maiores, e mesmo dispositivos que não os usem, devem não reconhecer erros quando os recebem. E também o dispositivo de mailbox, comentado acima. Esta projeto corresponde aos blocos cinza da Figura 1.

Será utilizada as ferramentas Simulink, Stateflow e HDL Coder, da MathWorks e ISE, da Xilinx.

As duas primeiras serão responsáveis pela modelagem de todo os módulos e máquinas de estado e suas interações entre si, além de permitir simulações comportamentais do sistema.

A ferramenta HDL Coder será utilizada para geração de código VHDL genérico, que pode ser utilizado em FPGA e ASICs.

E também será verificada a eficiência de um projeto utilizando ferramentas baseadas em modelos para geração automática de código. O Simulink e Stateflow serão utilizados com licenças acadêmicas, presentes nos Laboratórios de Pesquisa do Instituto de Informática, e para o HDL Coder, uma cooperação com o Fraunhofer IESE, para ser possível a geração de código. Estes módulos serão testados em placas da Xilinx, mas poderiam ser utilizados em qualquer FPGA ou ASIC, já que o código gerado é um VHDL genérico.

Tabela 1. Cronograma

Atividade	Fev	Mar	Abr	Mai	Jun
Estudo detalhado do protocolo	X				
Ambientação com o ambiente de desenvolvimento	X	X			

Implementação		X	X		
Testes			X	X	
Redação final do trabalho				X	X

O projeto já foi iniciado. Parte do Data Link Layer já foi implementado e VHDL já é possível de ser gerado.

4. Conclusão

Será mostrado no trabalho que ferramentas guiada a modelos são de grande utilidade na implementação de projetos de sistemas embarcados, pois dão a possibilidade além de uma simulação prévia do comportamento do sistema, mas como também é possível gerar código automaticamente, em várias linguagens alvo, neste caso VHDL, fazendo o projeto mais curto, entendível e menos suscetível a erros. Além de uma alta reusabilidade, já que é possível também gerar código para outras linguagens e plataformas, como C e microcontroladores.

Também será comparado o VHDL para o controlador CAN gerado com um Verilog já existente e encontrado na internet. Além da implementação de um módulo extra, que diminui a necessidade de processamento de módulos acoplados, e permite conexão mais simples entre controlador e aplicações.

Referencias

Reynari, L.M., Cucinotta, F., Serra, A. e Lavagno, L. (2001) “A Hardware/Software Co-design Flow and IP Library Based of Simulink”. In: Design Automation Conference, 2001. Proceedings, pág. 593-598.

Weiss, K. A., E. C. Ong and N. G. Leveson, 2003, "Reusable Specification Components for Model-Driven Development". In International Conference on System Engineering (INCOSE '03), Crystal City, Virginia USA.

Douglas C. Schmidt, "Model-Driven Engineering," IEEE Computer, pp. 25-31, February, 2006

Bran Selic. 2003. The Pragmatics of Model-Driven Development. IEEE Software. Volume 20, Issue 5 (September 2003), 19-25.

Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. 2004. Reuse of software in distributed embedded automotive systems. In Proceedings of the 4th ACM international conference on Embedded software (EMSOFT '04). ACM, New York, NY, USA, 203-210.

Robert Siegmund, Dietmar Müller, "A Novel Synthesis Technique for Communication Controller Hardware from declarative Data Communication Protocol Specifications," Design Automation Conference, p. 602, 39th Design Automation Conference (DAC'02), 2002

Van Beeck, K. and Heylen, F. and Meel, J. and Goedemé, T., “Comparative study of Model-based hardware design tools”, 2010

Vizhanyo, A. and Neema, E. and Kalmar, Z. and Shi, F. and Karsai, G., “Model-Driven Software Development of Model-Driven Tools: A Visually-Specified Code Generator for Simulink/Stateflow”, 2008

MathWorks, Stateflow® and Stateflow® Coder™ 7 User’s Guide

MathWorks, Simulink® 7 User’s Guide

MathWorks, Simulink® HDL Coder™ 2 User’s Guide

Carvalho, F.C., Jansch-Porto, I., Freitas, E.P., Pereira, C.E., “The TinyCAN: an optimized CAN controller IP for FPGA-based platforms”, Proceedings of 10th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2005, September 19-22, 2006, Catania, Italy