UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
ESPECIALIZAÇÃO EM BANCO DE DADOS

MARIANY APARECIDA DE SOUZA

GERAÇÃO AUTOMÁTICA DE CÓDIGO A PARTIR DA MANIPULAÇÃO DE MODELOS

MONOGRAFIA DE ESPECIALIZAÇÃO

PATO BRANCO
2017

MARIANY APARECIDA DE SOUZA

GERAÇÃO AUTOMÁTICA DE CÓDIGO A PARTIR DA MANIPULAÇÃO DE MODELOS

Trabalho de Conclusão de Curso, apresentado ao II Curso de Especialização em Banco de Dados, da Universidade Tecnológica Federal do Paraná, campus Pato Branco, como requisito parcial para obtenção do título de Especialista.

Orientador: Prof. Marcelo Teixeira.

PATO BRANCO
2017

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Pato Branco
Diretoria de Pesquisa e Pós-Graduação
II Curso de Especialização em Banco de Dados – Administração e Desenvolvimento

TERMO DE APROVAÇÃO

GERAÇÃO AUTOMÁTICA DE CÓDIGO A PARTIR DA MANIPULAÇÃO DE MODELOS

por

MARIANY APARECIDA DE SOUZA

Este Trabalho de Conclusão de Curso foi apresentado em 24 de fevereiro de 2017 como requisito parcial para a obtenção do título de Especialista em Banco de Dados. O(a) candidato(a) foi arguido(a) pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Marcelo Teixeira
Prof.(a) Orientador(a)

_____
Marco Antonio de Castro Barbosa
Membro titular

_____
Bruno Cesar Ribas
Membro titular

"O Termo de Aprovação assinado encontra-se na Coordenação do Curso"

AGRADECIMENTOS

A Deus por ter me dado força de vontade, persistência e coragem para enfrentar e superar as dificuldades.
Aos meus pais, pela motivação e apoio.
A universidade pela oportunidade de realização do curso.
Ao professor orientador pela confiança e apoio.
A todos os professores, meus eternos agradecimentos, por proporcionarem o conhecimento com dedicação e paciência.

Tudo o que um sonho precisa para ser realizado é alguém que acredite que ele possa ser realizado.

Roberto Shinyashiki

RESUMO

SOUZA, Mariany Aparecida. Geração automática de código a partir da manipulação de modelos. 2017. 42 f.. Monografia (II Curso de Especialização em Banco de Dados) - Universidade Tecnológica Federal do Paraná. Pato Branco, 2017.

Este trabalho baseia-se no processo de integração entre uma ferramenta de modelagem que utiliza máquinas de estados ou Autômatos, o Supremica, com o hardware. O Supremica é uma ferramenta que permite fazer modelagem de problemas usando máquinas de estados. Com ela pode-se também simular e implementar todas as operações necessárias para obter ou sintetizar o software. A modelagem é útil por ajudar a simplificar o entendimento do sistema e a resolver inúmeros problemas práticos como, por exemplo, problemas de controle industrial, coordenação de múltiplos equipamentos em fábricas, etc. Quando se faz o uso da modelagem, surgem muitas vantagens como: modularização, ou seja, poder olhar individualmente para cada componente e para cada requisito; formalização, que permite obter um software com garantias de qualidade, etc. Porém, mesmo com múltiplos recursos, o Supremica, assim como outras ferramentas de modelagem similares, não exporta o modelo para uma linguagem de programação compatível com o hardware. O Supremica oferece recurso para exportação em alguns formatos, como o html, dot e dentre eles o XML, mas não para linguagens como C, aceita pela maioria dos recursos de hardware. Portanto, a proposta deste trabalho objetiva solucionar este problema desenvolvendo uma aplicação capaz de ler um arquivo XML gerado pelo Supremica e convertê-lo na linguagem de programação, neste caso o C. Desta forma, o hardware pode interpretar e executar fielmente o que foi projetado na modelagem levando ao chão de fábrica os recursos de um sistema que foi concebido com alto nível de abstração e, teoricamente, de maneira mais clara que a usual.

Palavras-chave: Supremica. Máquinas de estados. XML. Linguagem C.

ABSTRACT

SOUZA, Mariany Aparecida. Automatic code generation from models manipulation. 2017. 42f. Monography (II Specialization Course in Database) - Federal University of Technology - Parana. Pato Branco, 2017.

This work is based on the process of integrating a modeling tool that uses states machines or Automata, the Supremica, with hardware programming. Supremica is a tool that allows to model problems using state machines. It is also possible to simulate and implement all the operations necessary to obtain or synthesize software. Modeling is useful in helping to simplify understanding of the system and solve numerous practical problems, such as problems of industrial control, coordination of multiple equipment in factories, etc. When using modeling, many advantages arise: modularization, that is, being able to look individually at each component and for each requirement; formalization, which allows to obtain a software with guarantees of quality, etc. However, even with multiple features, Supremica, as well as other similar modeling tools, does not export the model to a hardware-compatible programming language. Supremica offers export feature in some formats such as html, dot and among them XML, but not for languages like C, supported by most hardware devices. Therefore, the purpose of this work is to solve this problem by developing an application capable of reading an XML file generated by Supremica and converting it into the programming language, in this case the C. In this way, the hardware can interpret and execute faithfully what was designed, taking to the factory floor the features of a system that was designed with a high level of abstraction and, theoretically, more clearly than usual.

Keywords: Supremica. State-machines. XML. C language.

LISTA DE SIGLAS

ACLK	*Auxiliary Clock*
CLP	Controle Lógico Programável
CPU	*Central Processing Unit*
GIE	*General Interrupt Enable*
IDE	*Integrated Development Environment*
IEC	*International Engineering Consortium*
JVM	*Java Virtual Machine*
LCD	*Liquid Crystal Display*
MCLK	*Master Clock*
MSP	*Mixed-Signal Processor*
PC	*Program Control*
PCO	*Prioritized Composition Operator*
PWN	*Pulse Width Modulation*
RISC	*Reduced Instruction Set Computer*
SFC	*Sequencial Function Chart*
SMCLK	*Sub-Main Clock*
SP	*Stack Pointer*
TCS	Teoria de Controle Supervisório
UML	*Unified Modeling Language*
XML	*eXtensible Markup Language*

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE EQUAÇÕES

SUMÁRIO

1 INTRODUÇÃO

O presente trabalho está baseado em como integrar uma ferramenta de modelagem, o Supremica com o *hardware*, no caso um microcontrolador. Para este trabalho foi utilizado o hardware, MSP430, da *Texas Instruments*.

O Supremica é uma ferramenta que permite fazer modelagem de problemas usando máquinas de estados. É uma IDE com uma interface fácil de usar e serve para analisar, sintetizar e simular todas as operações necessárias para o *software*, partindo de versões do sistema obtidas com alto nível de abstração, concentrando-se no nível de eventos, e assim separar partes de seu comportamento que são irrelevantes para o fim a que se destina o sistema.

A modelagem com máquinas de estados serve para especificar inúmeros sistemas e requisitos para os quais é conveniente uma abordagem declarativa, diferentemente da maioria dos paradigmas de desenvolvimento de software. Por exemplo, em problemas de controle industrial, ou na coordenação de múltiplos equipamentos (como robôs) em fábricas, muitas vezes o núcleo do sistema de controle é apoiado sobre a sequencialização lógica dos componentes, de tal forma que o controle de parâmetros como tempo, força, velocidade, são, sob essa óptica, irrelevantes. Para esses casos, a modelagem por máquinas de estados ou autômatos é altamente conveniente.

As vantagens podem surgir ao se modelar um sistema via máquinas de estados. Por exemplo, pode-se olhar individualmente para cada componente e para cada requisito, o que simplifica a implementação do sistema como um todo. Ademais, este tipo de sistema, para microcontroladores, muitas vezes possuem o hardware sob medida e quando mal analisados tem como consequência prejuízos financeiros, problemas de segurança em chão de fábrica, de prazos estourados. Note, ainda, que nem sempre é possível parar uma fábrica inteira a fim de testar um sistema de controle. Normalmente, essa tarefa precisa ser feita via simulação e, ao ser implementado em hardware, deve-se ter a garantia (de preferência formal) de que o sistema funciona conforme o esperado. Salienta-se ainda que, em ambientes fabris, a produtividade de um sistema sob controle é uma questão de complexidade de mercado. Portanto, o sistema de controle pode ser alcançado por meio do uso de um ferramental que possibilitem explorar essas questões.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo geral é converter o modelo da máquina de estado, exportado em XML pelo Supremica, para a linguagem C.

1.2.2 Objetivos Específicos

Os objetivos específicos serão os especificados abaixo:

- Realizar estudo da ferramenta Supremica;
- Descrever e analisar a estrutura do arquivo XML gerado pelo Supremica;
- Desenvolver uma aplicação em Java para interpretar o XML;
- Converter, através da aplicação, o código para linguagem C;

1.3 JUSTIFICATIVA

Apesar do Supremica ser uma ferramenta com muitos recursos (modelagem, verificação, simulação, operações, síntese, etc.) para a modelagem e a manipulação de máquinas de estados, infelizmente, assim como outras ferramentas similares, não oferece uma forma de transformar o modelo criado em uma linguagem de programação, que permita ser interpretada pelo microcontrolador. O que é possível fazer no Supremica é a exportação do modelo para alguns tipos de formatos, tais como o XML, *html*, *dot*, que contém uma estrutura com os estados, transições e eventos que foram definidos na modelagem.

O Supremica tem como estrutura básica o Autômato Finito (ou máquina de estado finito) e que em conjunto conseguem definir o problema principal (MENEZES, 2000).

Os microcontroladores podem interpretar os modelos de Autômatos por meio de linguagens de programação, e a linguagem C é uma linguagem de alto nível, altamente portátil, uma vez o código implementado para um determinado chip pode ser adaptado para funcionar em outros chips e a arquitetura dos microcontroladores favorece a utilização desta linguagem (PEREIRA, 2005).

A proposta do trabalho é apresentar uma solução para essa inconveniência, de forma que o modelo exportado, seja lido e tratado por meio de uma aplicação, levando a um outro

formato de exportação. A aplicação desenvolvida na linguagem Java e irá ler e tratar o XML como dado de entrada, manipular as informações e gerar a saída de um arquivo em C.

A transformação do modelo contido no arquivo em XML para outra linguagem, tenta otimizar o trabalho de pessoas que fazem o uso do Supremica, tanto em nível acadêmico quanto industrial. Atualmente é preciso ler o modelo em XML e implementá-lo novamente com uma segunda ferramenta, simplesmente para obter o arquivo em C. Essa tarefa é manual, ao passo que um modelo para um problema real pode conter milhares de estados e transições, inviabilizando a abordagem na prática. Além disso, existem recursos avançados de controle industrial embutidos no Supremica, como síntese com variáveis, cuja aplicação prática na indústria se torna inviável devido à carência que se tem na geração de código.

1.4 ESTRUTURA DO TRABALHO

O trabalho está organizado da seguinte forma: no capítulo 2 está uma breve abordagem sobre os Autômatos Finitos, o capítulo 3 apresenta o Supremica e suas características, o capítulo 4 aborda os microcontroladores MSP430, o capítulo 5 apresenta o desenvolvimento da aplicação e por fim no capítulo 6 onde serão apresentadas as conclusões e os resultados finais.

2 AUTÔMATO FINITO

O seguinte capítulo está baseado na obra de Menezes (2000)

Um Autômato finito pode ser visto como uma máquina composta por fita, unidade de controle e função de transição.

A fita é um dispositivo de entrada que contém a informação a ser processada, é finita, dividida em células, onde cada uma armazena um símbolo que pertencem a um alfabeto de entrada. A unidade de controle reflete o estado corrente da máquina, possui uma unidade de leitura a qual acessa uma célula da fita de cada vez, também possui um número finito e predefinido de estados e, lê o símbolo de uma célula de cada vez. E por fim, a função de transição é quem comanda as leituras e define o estado da máquina, é uma função parcial que, dependendo do estado corrente e do símbolo lido, determina o novo estado do Autômato. O Autômato finito não possui memória de trabalho, portanto, para armazenar as informações passadas necessárias ao processamento, deve-se usar o conceito de estado. (MENEZES, 2000)

Um Autômato Finito é representado por M e é uma 5-tupla, como mostrado na Equação abaixo.

$$\mathrm{M} = (\sum, Q, \delta, q0, F)$$

Equação 1 – Representação de um Autômato

Onde a fórmula representa:

Σ é o alfabeto de símbolos de entrada;

Q é o conjunto de estados possíveis do Autômato;

δ é a função programa ou função parcial ou relação de transição;

q0 é o estado inicial tal que q0 é elemento de Q;

F é o conjunto de estados finais tal que F está contido em Q;

O processamento de um Autômato finito M, para entrada em w, consiste na aplicação da função programa, repetidamente, para cada símbolo de w até que ocorra uma condição de parada (MENEZES, 2000).

Para demonstrar, considere o Autômato abaixo, da Figura 1 e 2 como exemplo:

$$M_1 = \left(\{a, b\}, \{q_0, q_1, q_2, q_f\}, \delta_1, q_0, \{q_f\}\right)$$

Equação 2 - Exemplo de Autômato Finito

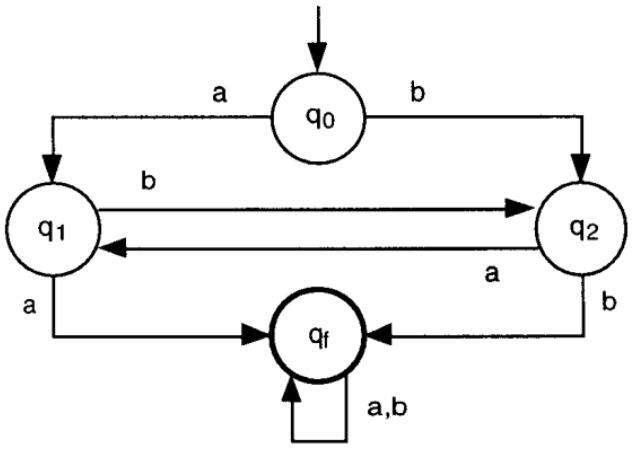

Figura 1 - Representação do autômato.

O algoritmo usa os estados representados por q1 e q2 para armazenar o símbolo anterior. Após identificar dois a ou dois b consecutivos, o Autômato passa para o estado final qf, somente para terminar o processamento.

Um Autômato Finito sempre termina ao processar qualquer entrada, não existe a possibilidade de ciclo infinito. A parada de um processamento pode ser aceitando ou rejeitando uma entrada w.

As condições de parada são as seguintes:

a) Após processar o último símbolo da fita, o Autômato Finito assume um estado final: o Autômato para e a entrada w é aceita;
b) Após processar o último símbolo da fita, Autômato Finito assume um estado não-final, o Autômato para e a entrada w é rejeitada;
c) A função programa é indefinida para o argumento (estado corrente e símbolo lido); a máquina para e a palavra de entrada w é rejeitada;

2.1 AUTÔMATOS FINITOS COM SAÍDA

Para Menezes (2000, p.72), “o conceito básico de Autômato Finito possui aplicações restritas, pois a informação de saída é limitada à lógica binária aceita/rejeita”. As saídas podem estar relacionadas com as transições (Máquina de Mealy) ou com os estados (Máquina de Moore).

Ambas as máquinas, a saída não pode ser usada como memória auxiliar, como segue abaixo:

a) É definida sobre um alfabeto especial, denominado Alfabeto de Saída;
b) A saída é armazenada em uma fita independente da de entrada;
c) A cabeça da fita de saída move uma célula para direita a cada símbolo gravado;
d) O resultado do processamento do Autômato Finito é o seu estado final e a informação contida na fita de saída;

2.1.1 Máquina de Mealy

De acordo com Menezes (2000, p. 73), “a Máquina de Mealy é um Autômato Finito modificado de forma a gerar uma palavra de saída para cada transição”.

Uma Máquina de Mealy M é um Autômato Finito com saídas associadas às transições. É representada por uma 6-tupla, mostrado na Equação abaixo:

$$\mathrm{M} = (\sum, Q, \delta, q0, F, \Delta)$$

Equação 3 - Representação máquina de Mealy

Onde a fórmula é representada por:

Σ é alfabeto de símbolos de entrada;

Q é o conjunto de estados possíveis do Autômato o qual é finito;

δ é a função programa ou função de transição;

q0 é o estado inicial do Autômato tal que q0 é elemento de Q;

F é o conjunto de estados finais tal que F está contido em Q;

Δ é o alfabeto de símbolos de saída;

O processamento de uma Máquina de Mealy, consiste na sucessiva aplicação da função programa, repetidamente até ocorrer uma condição de parada. Se todas as transições resultam em uma saída vazia, então a Máquina de Mealy processa como se fosse um Autômato Finito.

2.1.2 Máquina de Moore

Segundo Menezes (2000, p. 74), ”uma Máquina de Moore M é um Autômato Finito Determinístico com saídas associadas aos estados”. É representada por um 7-tupla, como apresentada na Figura 5:

$$M = (\sum, Q, \delta, q0, F, \Delta, \delta s)$$

Equação 4 - Representação da máquina de Moore

Onde a fórmula é:

Σ é o alfabeto de símbolos de entrada;

Q é o conjunto de estados possíveis do Autômato o qual é finito;

Δ é a função programa ou a função de transição;

q0 é o estado inicial tal que q0 é elemento de Q;

F é o conjunto de estados finais tal que F está contido em Q;

Δ é o alfabeto de símbolos de saída;

δ0 função de saída;

O processamento de uma Máquina de Moore, consiste na sucessiva aplicação da função programa para cada símbolo de w até ocorrer uma condição de parada. Se todos os estados resultam em uma saída vazia, então a Máquina de Moore processa como se fosse um Autômato Finito (MENEZES, 2000).

3 SUPREMICA

Supremica é uma ferramenta de modelagem e manipulação de modelos que faz o uso de autômatos para modelar problemas. Serve para verificação automática e síntese de controladores para Sistemas de Eventos Discretos (SED) (AKESSON, 2012).

Um SED consiste em um sistema de estados discretos dirigidos por eventos, onde sua evolução de estados depende da ocorrência de eventos discretos assíncronos no tempo (CASSANDRAS e LAFORTUNE, 2008).

A Teoria de Controle Supervisório (TCS) consiste em um formalismo para modelagem e controle de SEDs utilizando linguagens formais e autômatos (RAMADGE e WONHAM, 1989).

Com a TCS é possível resolver alguns dos problemas relacionados com os altos requisitos de flexibilidade em muitos novos sistemas de produção. Novas instalações de produção devem ser fáceis de reconfigurar, possibilitar a adição e remoção de recursos e também ser capazes de produzir uma variedade de produtos (RAMADGE e WONHAM, 1989).

Quando os sistemas de produção ou os produtos mudam com frequência, pode ser interessante ou até necessário sintetizar automaticamente o código de controle (AKESSON, 2006).

Segundo Akesson (2006, p. 1), “para lidar com a verificação e síntese de problemas de tamanho industrial, o Supremica implementa algoritmos que explora a estrutura modular do problema, juntamente com métodos simbólicos, para representar eficientemente grandes espaços de estado”.

De acordo com Akesson (2006), “muita teoria tem sido desenvolvida dentro da estrutura do TCS, porém poucas as aplicações industriais da teoria foram apresentadas até agora. Várias ferramentas foram criadas para experimentar os conceitos introduzidos no TCS, mas a maioria delas concentra-se no uso acadêmico em vez do industrial. ” E o Supremica possui suporte para executar o supervisor a partir de uma simulação gráfica do ambiente. Essa ideia possui a intenção de melhorar as opções de ferramentas existentes na literatura, como:

a) TCT e GRAIL: é uma ferramenta da TCS com interfaces primitivas baseadas em texto.
b) UMDES: é uma biblioteca de rotina C para TCS. .
c) UKDES: contém os algoritmos básicos e tem uma interface gráfica de usuário simples.

d) J-DES: possui uma interface gráfica com o usuário e oferece funcionalidade semelhante à TCT e UKDES.
e) VALID: é uma ferramenta conhecida comercialmente para a teoria de controle de supervisão. Foi desenvolvido pela *Siemens Corporate Research*, na Alemanha. Aparenta ser a ferramenta mais completa entre as ferramentas de supervisão que existem, mas o foco está na verificação e não na síntese.

Porém essas ferramentas se mostram ineficientes pelo menos em um dos seguintes aspectos: facilidade de uso, verificação, síntese, capacidade de lidar com grandes sistemas, simulação, geração de código e execução de código, etc. (AKESSON, 2004).

3.1 AUTÔMATOS E O SUPREMICA

De acordo com Akesson (2006), "o projeto do Supremica consiste em vários Autômatos que juntos definem o problema principal. O comportamento de todos os Autômatos é definido pelo operador de composição priorizado (PCO) ".

O Supremica tem como estrutura básica os Autômatos Finitos Determinísticos. Como exposto no capítulo anterior, o Autômato possui um conjunto de estados, transições entre os estados e os eventos (AKESSON, 2006).

O PCO serve para poder expressar tanto a composição síncrona total quanto a composição de difusão usada, por exemplo, por diagramas de estado em *Unified Modeling Language* (UML). PCO pode também misturar composições completa e de difusão.

> Akesson (2006, p.2) cada Autômato também tem tipo, pode ser de planta, especificação ou supervisor. A composição síncrona priorizada dos Autômatos da planta define o comportamento da planta. O termo sistema usados frequentemente na sequência é usado para denotar a composição síncrona priorizada de todos os Autômatos no projeto.

O Supremica também permite sintetizar supervisores não-bloqueadores (bloqueantes) e controláveis (RAMADGE e WONHAM, 1989). Foram desenvolvidos algoritmos eficientes que exploram a modularidade do sistema, para verificação e síntese de supervisores controláveis, um algoritmo eficiente para verificação de inclusão de linguagem. Também implementa a minimização do estado de um Autômato.

Supremica também permite que o usuário limite ou projete eventos do alfabeto em um Autômato, implementa também algoritmos para controlabilidade limitada que são úteis na supervisão híbrida computador-humano de sistemas de eventos discretos (AKESSON, 2006).

É melhor para o usuário quando o número de estados é pequeno, todavia, se o Autômato contém muitos estados ou muitas transições, uma apresentação gráfica torna-se facilmente desordenada. Para esta situação o Supremica possui um explorador que exibe um único estado e o conjunto de eventos habilitados nesse estado. O usuário pode clicar em um evento para executar esse evento e, como resultado, alterar o estado (AKESSON, 2006).

3.1.1 Síntese

De acordo com Akesson (2006, p. 2) “O Supremica implementa algoritmos de síntese monolítica para resolver problemas de não-bloqueio, controlabilidade e combinação de não-bloqueio e controlabilidade”.

A abordagem monolítica de síntese de supervisor para SEDs é também chamada de Abordagem RW (RAMADGE e WONHAM, 1989).

O Supremica implementa algoritmos modulares para verificação e síntese dos problemas de controlabilidade e que foram avaliados e capazes de sintetizar um conjunto de supervisores interagindo com tempo muito limitado e requisitos de memória (AKESSON, 2006).

3.1.2 Verificação

Verificação busca comprovar as propriedades do sistema por ajuda de métodos formais. Como a ferramenta Supremica contém seus próprios algoritmos de verificação, então pode-se fazer verificação de controlabilidade, verificar problemas de tamanho quase arbitrário e para verificar propriedades não-bloqueando, é usado uma aproximação da força-bruta e está consequentemente incapaz de segurar sistemas com mais de alguns milhões de estados (AKESSON, 2006).

Vale ressaltar que o processo de síntese, quando conduzido sobre certa propriedade (como controlabilidade, por exemplo) descarta a necessidade de verificação a posteriori. Por essa razão, o foco desse trabalho está mais concentrado na TCS e seu produto final, o supervisor, e em como ele pode ser traduzido em código implementável. Dessa forma, note que o processo de verificação já estaria embutido em nossos resultados.

3.1.3 Simulação

Uma interface para a simulação gráfica foi desenvolvida. Isso torna possível gravar animações gráficas e carregá-las em conjunto com os Autômatos de estado. Supremica pode então computar supervisores e executá-los contra a animação gráfica, como mostra a figura 3.

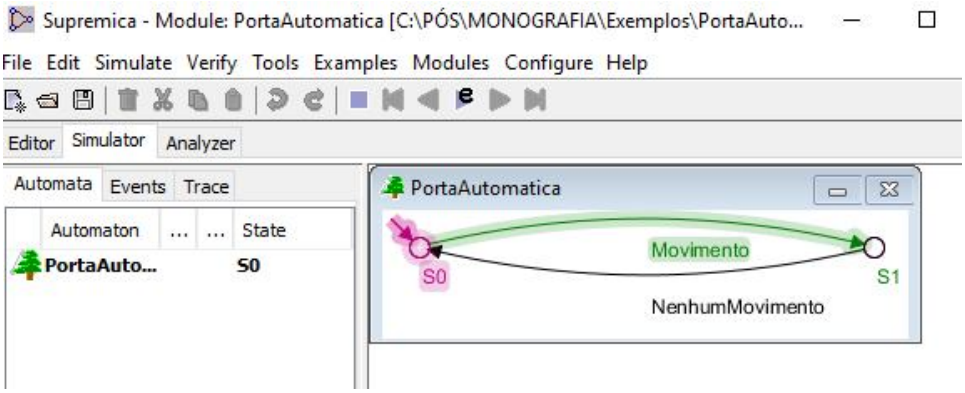

Figura 2 - Interface de simulação

Mas a principal vantagem de dispor de um simulador é para que um problema possa ser modelado e testado incrementalmente, em tempo real. Na prática isso elimina a maioria dos erros de modelagem que seriam transferidos para as etapas posteriores de projeto.

4 MICROCONTROLADORES MSP430

Neste Capítulo será apresentado alguns conceitos e características dos Microcontroladores MSP430.

Os MSP430 da *Texas Instruments* são de baixa potência, baseados em arquitetura RISC, e possui processadores de sinais mistos, inclui recursos avançados como consumo de energia RF incorporado e de segurança, como criptografia AES (*Texas Instruments*, 2006).

4.1 ARQUITETURA MSP430

Nesta seção é apresentado algumas das principais características dos *chips* MSP430, uma breve descrição do funcionamento da sua CPU (*Central Processing Unit*), as modalidades de operação e a organização da memória.

4.1.1 CPU

Para Pereira (2005, p.23), "a arquitetura RISC combina um conjunto reduzido de instruções com uma arquitetura de barramento clássico Von Neumann, permitindo que a CPU possua um espaço único de endereçamento de memória".

É importante saber que a CPU desses *chips* possui três barramentos: endereços, dados e controle.

Possuindo largura de 16 bits, o barramento de endereços permite acessar até 65.536 posições de memória. Já com relação ao barramento de dados, a largura de 16 *bits* é possível que a CPU possa processar informações em lotes de 16 *bits* (PEREIRA, 2005).

A CPU possui registradores internos, como apresentado na figura 6, todos de 16 *bits*, nomeados de R0 a R15. Segundo Pereira (2005, p.24), "os quatro primeiros registradores possuem finalidades dedicadas e via de regra não podem ser utilizados para outros propósitos. Os demais registradores de R4 A R15 podem ser utilizados para propósitos gerais".

De acordo com Pereira (2005, p.24), "os MSP430 não possuem um registrador acumulador específico, qualquer um dos 16 registradores da CPU pode funcionar como fonte ou destino de uma operação". Segue abaixo a Figura 4 com a lista de registradores.

Registrador	Nome/Função
R0	Contador de programa (PC)
R1	Apontador da pilha (SP)
R2	*Status*/Gerador de constantes 1
R3	Gerador de constantes 2
R4	Registrador de propósito geral (GPR)
R5	Registrador de propósito geral (GPR)
R6	Registrador de propósito geral (GPR)
R7	Registrador de propósito geral (GPR)
R8	Registrador de propósito geral (GPR)
R9	Registrador de propósito geral (GPR)
R10	Registrador de propósito geral (GPR)
R11	Registrador de propósito geral (GPR)
R12	Registrador de propósito geral (GPR)
R13	Registrador de propósito geral (GPR)
R14	Registrador de propósito geral (GPR)
R15	Registrador de propósito geral (GPR)

Figura 3 - Registradores

O R0 é o contador de programa (PC – *Program Control*), visa apontar a próxima instrução a ser lida pela memória e executada pela CPU. O contador de programa nesta arquitetura pode ser lido e escrito diretamente pelo *software* em execução, permitindo o uso de técnicas, como, o desvio calculado, por exemplo (PEREIRA, 2005).

De acordo com Pereira (2005, p.24), "o R1 é o Apontador da Pilha (SP – *Stack Pointer*), é utilizado para indicar à CPU a localização do topo da pilha de memória". Utiliza-se a pilha de memória para o armazenamento de endereços de retorno nas chamadas de sub-rotinas e tratamento de interrupções.

O R2 é o Registrador SR/CG1, possui duas funções: pode funcionar como o registrador de estado da CPU, ou como um gerador de constantes (PEREIRA, 2005).

Para Pereira (2005, p. 25), "o registrador de estados (SR) possui o propósito de armazenar *bits* de estados e de controle da CPU".

Os Registradores Geradores de Constantes (R2 e R3) possuem a função muito importante, são responsáveis pela geração de constantes numéricas necessárias à emulação de instruções (PEREIRA, 2005).

Os demais, os registradores de propósito geral (R4 a R15) podem ser utilizados para funções diversas à escolha do usuário, como armazenamento de variáveis de uso intensivo, apontadores de endereço.

E por fim, a organização da memória, onde o espaço total de endereçamento é de 64Kbytes. Cada posição de memória é formada por um byte e a CPU pode endereçar bytes individualmente (PEREIRA, 2005).

De acordo com Pereira (2005, p.26), “um aspecto importante do mapa de memória desta arquitetura está na separação que existe entre as regiões da memória”.

4.2 SISTEMA DE INTERRUPÇÃO

Para Pereira (2005, p. 107), “uma interrupção consiste em um evento externo ao programa que provoca um desvio no seu fluxo, de forma que a CPU possa executar um subprograma em resposta ao evento. Ao término desse subprograma, o fluxo do programa retorna ao ponto em que se encontrava antes da interrupção”.

A importância das interrupções está quando há necessidade da CPU responder rapidamente a um evento, mas sem a perda de capacidade de executar outras operações enquanto ele não ocorra (PEREIRA, 2005).

4.3 OSCILADOR

Para Pereira (2005, p. 116), “os microcontroladores MSP430 incluem um avançado sistema de *clock*, que permite a operação da CPU e dos periféricos a partir de fontes diferentes”.

Os sinais vindos dos osciladores podem ser selecionados para a geração de três sinais de *clock* do sistema:

a) O *clock* principal (MCLK – *Master Clock*) é utilizado para sincronizar a CPU e eventualmente outros periféricos.
b) O *clock* secundário (SMCLK – *Sub-Main Clock*) é utilizado como fonte de *clock* alternativa para os diversos periféricos do microcontrolador.
c) O *clock* auxiliar (ACLK – *Auxiliary Clock*) que funciona como fonte de *clock* de precisão para periféricos durante modos de baixo consumo.

4.4 PORTAS DE E/S

Para Pereira (2005, p. 138), “os MSP430 podem ter até seis portas de entrada/saída de uso geral, cada uma com até 8 pinos, onde cada pino pode ser configurado individualmente para funcionar como uma entrada/saída”.

Cada porta tem um conjunto de registradores que controlam a sua operação:

a) PxIN: é um registrador de entrada para a leitura do estado dos pinos da porta;
b) PxOUT: é um registrador de saída para a escrita de valores nos pinos da porta;
c) PxDIR: é um registrador para o controle individual da direção dos pinos da porta, se entrada ou saída;
d) PxSEL: é um registrador para seleção da função do pino, que seleciona entre a função de E/S normal e a função eventualmente multiplexada ao pino;

4.5 TIMERS

Entre os *timers*, temos o *Timer* A e o *Timer* B. O *timer* A é um contador/temporizador de 16 *bits* e o *timer* B pode ser considerado uma evolução do *timer* A. (PEREIRA, 2005)

4.5.1 *Timer* A

Algumas das características do *Timer* A são:

a) Um contador assíncrono progressivo/regressivo de 16 *bits* com módulo programável e capacidade de interrupção;
b) Capacidade de operar a partir de diversas fontes de *clock* interno e externo;
c) Até três registradores de CCP - captura, comparação e PWM.
d) Capacidade de captura originada a partir da saída do comparador analógico;

Para Pereira (2005, p. 145), “o modo de captura pode ser utilizado para medição de período de sinais ou outras medições de tempo em que se requeiram máxima precisão e mínima intervenção da CPU”.

Segundo Pereira (2005, p.146) “o modo de comparação/PWM (*Pulse Width Modulation*) que pode ser utilizado para geração de pulsos ou interrupções com intervalos de tempo precisos ou também para a geração de sinais PWN”.

4.5.2 *Timer* B

O *timer* B possui as funcionalidades do *timer* A e mais algumas outras características:

a) Capacidade de configurar o contador principal para larguras diferentes.
b) Três ou sete canais de Captura, Comparação e PWM.
c) Capacidade de agrupamento de canais.
d) Capacidade de colocar as saídas do *timer* em estado de alta impedância por meio de um sinal de controle externo.

4.6 TEMPORIZADOR BÁSICO

De acordo com Pereira (2005, p. 166), "o temporizador básico é formado por dois contadores de 8 *bits* destinados a aplicações distintas".

O primeiro contador (BTCNT1) é aplicado para gerar o sinal de *clock* que é necessário para o módulo controlador de LCD (*Liquid Crystal Display*). O segundo contador (BTCNT2) pode ser aplicado como divisor de frequências programável e com capacidade de gerar interrupções, sendo que a sua principal aplicação é a geração de interrupções periódicas da CPU, e pode funcionar como base de um sistema de relógio de tempo real.

4.7 WATCHDOG

Para Pereira (2005, p. 311), "o *watchdog* é um temporizador projetado com a finalidade de vigiar a execução do *software*. Consiste basicamente em um contador de 16 *bits* que, ao final do período de contagem, pode provocar um *reset* da CPU".

É possível programar o *watchdog* somente gerar uma interrupção e passar a funcionar como um temporizador. Desta forma, o *watchdog* pode operar para executar o *reset* e também como um temporizador (PEREIRA, 2005).

A base de operação do *watchdog* é um contador de 16 *bits* cuja fonte de *clock* pode ser selecionada entre duas opções: o sinal de *clock* secundário ou o sinal de *clock* auxiliar. A seleção é feita pelo *bit* WDTSSEL, localizado no registrador WDTCTL.

Outro fator importante em relação ao registrador de controle do *watchdog*, o WDTCTL, é protegido por senha, ou seja, a escrita é controlada pelo *hardware* do microcontrolador.

É possível desligar o módulo de forma a economizar energia, o que pode ser feito pelo *bit* WDTHOLD.

5 CONVERTERC

Apresenta-se neste capítulo o caminho para o desenvolvimento da aplicação que fará a conversão do modelo que está em XML para a linguagem C.

Foi construído um exemplo no Supremica, que abrange a modelagem do problema, a exportação do modelo para o XML e a implementação do sistema de conversão.

5.1 MODELO NO SUPREMICA

Apresenta-se nesta seção um modelo de exemplo no Supremica. A Figura 5 mostra o exemplo de modelagem de uma porta automática.

Em uma situação real, a porta automática se comporta de forma que sem movimento permanece fechada e abre quando há movimento.

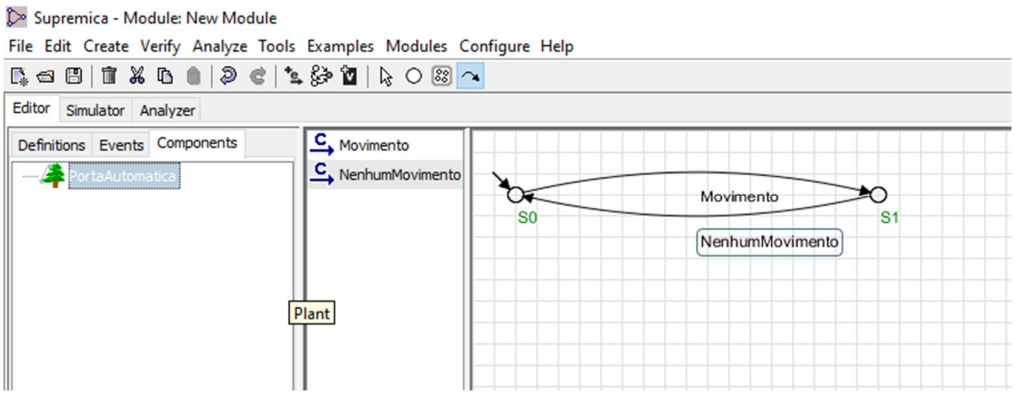

Figura 4 - Exemplo da porta automática

Para criar a modelagem é preciso incluir dois estados, o S0 e S1 que representam os estados fechado (S0) e aberto (S1) respectivamente, sendo o estado S0 o estado inicial.

Para controlar os estados, dois eventos foram criados, o “Movimento” e o “NenhumMovimento”, um para cada estado. O “Movimento” é acionado quando algum movimento é detectado, caso contrário, permanece no estado “NenhumMovimento”.

As transições são executadas quando um dos eventos, “Movimento” ou “NenhumMovimento” é acionado.

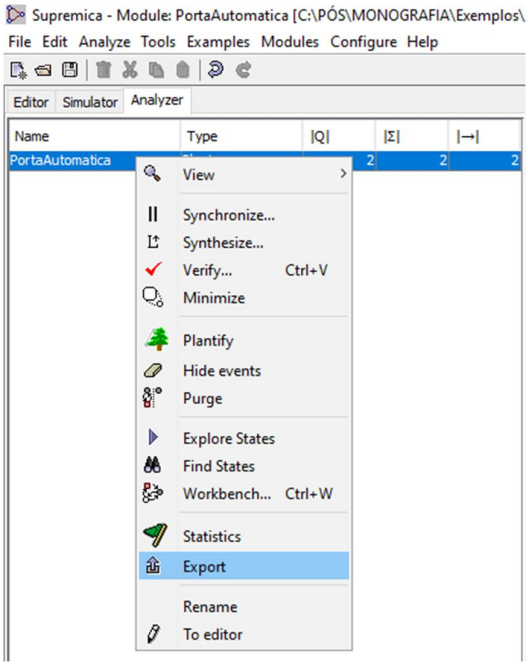

Figura 5 - Aba *analyzer*

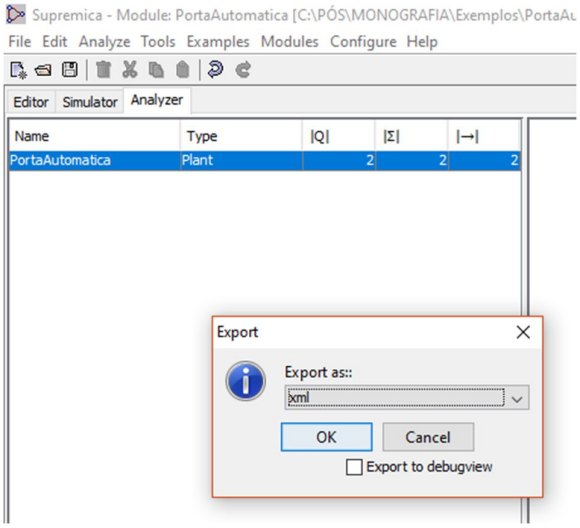

Figura 6 - Exportação para XML

Pela aba *Analyzer*, é possível fazer a exportação do modelo para XML, como mostra a Figura 6 e 7. O Supremica ainda vai pedir para indicar o local e o nome do arquivo antes de salvar.

O arquivo XML, mostrado na Figura 8, armazena a estrutura do modelo criado. Possui os estados, os eventos e as transições. O arquivo consiste em três subestruturas, *Events* com os

eventos, *States* com os estados e *Transitions* que são as transições criadas. Se iniciar a leitura do arquivo pela subestrutura <*Transitions*> pode-se perceber que o atributo *source* representa o estado inicial, o atributo *dest* representa o estado de destino e o atributo *event* representa o evento que foi acionado para a transição correspondente.

PortaAutomatica.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <Automata name="Untitled" major="0" minor="9">
3  <Automaton name="PortaAutomatica" type="Plant">
4      <Events>
5          <Event id="0" label="Movimento"/>
6          <Event id="1" label="NenhumMovimento"/>
7      </Events>
8      <States>
9          <State id="0" name="S0" initial="true" accepting="true"/>
10         <State id="1" name="S1" accepting="true"/>
11     </States>
12     <Transitions>
13         <Transition source="0" dest="1" event="0"/>
14         <Transition source="1" dest="0" event="1"/>
15     </Transitions>
16 </Automaton>
17 </Automata>
18
```

Figura 7 - Estrutura do modelo em XML

5.2 DESENVOLVIMENTO DA APLICAÇÃO

Nesta seção é abordado o processo de desenvolvimento da aplicação que tem por objetivo transformar essa estrutura do XML que será a entrada em um código C.

A aplicação foi desenvolvida na linguagem Java com a ferramenta eclipse e o código está disponível em https://github.com/maryapssouza/ConverterC.

Em um primeiro momento, a aplicação precisa realizar a leitura do XML e convertê-lo em objeto para facilitar a manipulação. Com a biblioteca *Xtream* pode-se otimizar a conversão do XML em objeto, pois essa biblioteca permite a serialização e deserialização de objetos utilizando representação em XML, como exposto na Listagem 1.

```
private Automata getXMLtoObject(){
        automata = FuncoesXML.converterXMLtOObject(xml);
        return automata;
}

public static Automata converterXMLtOObject(String arqxml){
        try {
                BufferedReader br = new BufferedReader(new FileReader(new File(arqxml)));
                String line;
                StringBuilder sb = new StringBuilder();

                while((line=br.readLine())!= null){
                        sb.append(line.trim());
                }
```

```
                String xml = sb.toString();
                XStream stream = new XStream(new DomDriver());
                stream.autodetectAnnotations(true);
                stream.alias("Automata", Automata.class);

                return (Automata) stream.fromXML(xml);
        } catch (Exception e) {
                Log.getLogger().log( Level.SEVERE, e.toString(), e);
                return null;
        }
}
```

Listagem 1 – Conversão do XML no objeto Autômato

O próximo passo é manipular as transições de forma que possa utilizá-las para informar as variáveis iniciais e as constantes. O tratamento das transições é mostrado na Listagem 2. Para pegar o número de transições e o número de estados, basta pegar o tamanho da lista de transições por meio da função *size*. Então, percorrendo a lista de estados, definimos o estado inicial que também precisa ser definido logo no início, pois as transições serão executadas a partir desse estado. Após isso, é preciso percorrer a lista de transições e pegar cada uma das posições de execução e armazená-las em um vetor, que será utilizado posteriormente.

```
private void setTransicoes(){

        Boolean percorre = false;
        if (!automata.getAutomatos().isEmpty()) { // if 1
                if (automata.getAutomatos().size() > 0) { // if 2

                        for (Automatos aut : automata.getAutomatos()) { // for 1
                                NTRANS = aut.getTransicoes().size();
                                NESTADOS = aut.getEstados().size();

                                for(Estados e : aut.getEstados()){
                                if (e.getInitial() != null) {
                                        if (e.getInitial()) {
                                                estadoInicial = e.getId();
                                                }
                                        }
                                }
                                for(Transicoes t : aut.getTransicoes()){

                                        Transicoes trans = new Transicoes();
                                        if(t.getSource().equals(estadoInicial)  &&  !percorre)  {

                                                trans = new Transicoes();

                                                trans.setPosicao(posicao);
                                                trans.setSource(t.getSource());
                                                trans.setDest(t.getDest());
                                                trans.setEvent(t.getEvent());
                                                vectorTransicoes.addElement(trans);
                                                percorre = true;
                                                continue;
                                        }
                                        if (percorre) {
                                                if (!vectorTransicoes.contains(t.getEvent())) {

                                                        posicao++;
                                                        trans.setPosicao(posicao);
                                                        trans.setSource(t.getSource());
                                                        trans.setDest(t.getDest());
                                                        trans.setEvent(t.getEvent());
                                                        vectorTransicoes.addElement(trans);
```

```
                                                }
                                        }
                                }
                        }       // fim for 1
                } // fim if 2
        } // fim if 1
}
```

Listagem 2 – Método de tratamento das transições

Definido a ordem de execução das transições, pode-se iniciar a conversão do XML.

A conversão é feita pelo método *converterJavaToC*, onde começa pelo cabeçalho e vai até a configuração do temporizador do timer A para o microcontrolador, mostrado na Listagem 3.

```
public void converterJavaToC() {
        setCabecalho();
        setConstantes();
        setVariaveis();
        setMapeiaEventosNaoControlaveis();
        setEventosSaida();
        setFuncoesInicializacao();
        setVariaveisGlobais();
        setMain();
        setTratamentoInterrupcao();
        setConfig_clk();
        setConfig_timer();
        setConfig_io();
        setConfig_tempo_ms();
}
```

Listagem 3 – Métodos de conversão

Na Listagem 4, basicamente, o sistema cria um arquivo .c com o método *writeContent* e vai alimentando-o enquanto vai passando pelos outros métodos.

O primeiro método *setCabecalho*, inclui a biblioteca *msp430.h*, que permite o acesso aos recursos de desenvolvimento voltado ao microcontrolador. Já o método *setConstantes*, define as constantes do sistema: NTRANS e NESTADOS com os dados do Autômato contido no XML.

```
private void writeContent(String wcontent){

        File file = new File(PATHFILE);
        try {
                if (!file.exists()) {
                        file.createNewFile();
                }
                FileWriter fw = new FileWriter(file.getAbsoluteFile());
                BufferedWriter bw = new BufferedWriter(fw);
                bw.write(wcontent);
                bw.close();
        } catch (IOException e) {
                e.printStackTrace();
        }
}
```

```
private void setCabecalho() {

	content = "#include \"msp430.h\" \n\n";
	writeContent(content);

}


private void setConstantes() {

	content += "#define NTRANS " + ntrans + " \n";
	content += "#define NESTADOS " + nestados + " \n";
	content += "#define BUFFER 10 \n\n";

	writeContent(content);

}
```

Listagem 4 – Métodos do sistema

O método *setVariaveis* serve para determinar os vetores *event, in_state, rfisrt* e *rnext*. O vetor *event* representa um vetor de eventos com base no número de transições. O *in_state* representa o vetor de estados, também com base no número de transições. O *rfisrt* representa um vetor de posições, onde os valores desse vetor indicam a posição, no vetor *event*, do último evento possível de ocorrer em um estado. E o *rnext* é um vetor que aponta as posições do vetor *events*, só que neste caso, a posição apontada é a do próximo evento possível de ocorrer no estado atual. Desta forma o algoritmo em C, consegue percorrer todos os eventos possíveis do estado até encontrar o evento ocorrido e realizar a transição.

No código exposto na Listagem 5, é simples, como já tem-se um vetor de transições carregado no método *setTransicoes*, então é necessário apenas, percorrer o vetor de transições e ir informando o evento para a variável *events* e o estado de destino informado para a variável *in_states* e desta forma monta-se os *arrays* de *events* e *in_states*, respectivamente.

```
private void setVariaveis() {
	String events = "";
	String in_states = "";
	String rfirst = "";
	String rnext = "";

	String source_aux ="";
	String source_aux2 = "";
	int iteracoes = 0;

	for(Transicoes t : vectorTransicoes){
		if (events == "") {
			events += t.getEvent();
		}else	{
			events += ", " + t.getEvent();
		}

		if (in_states == "") {
			in_states += t.getDest();
		}else	{
			in_states += ", " + t.getDest();
		}
```

```
                if (source_aux == "" || source_aux != t.getSource()) {
                        int posicao = contem(t.getSource());
                if (rfirst == "") {
                                rfirst += posicao;
                }       else{
                                rfirst += ", " + posicao;
                        }
                        if (iteracoes == vectorTransicoes.size()-1) {
                                rfirst += ", 0";
                        }
                        source_aux = t.getSource();
                }

                if  ((contem2(t.getSource())  >  1  &&  iteracoes  ==  0)  ||  (source_aux2  !=
t.getSource())) {
                        if (rnext == "") {
                                rnext += "0";
                        } else {
                                rnext += ", 0";
                        }
                } else {
                        if (rnext == "") {
                                rnext += contem(t.getSource()) - 1;
                        } else {
                                rnext += ", " + (contem(t.getSource()) - 1);
                        }
                }
                source_aux2 = t.getSource();
                iteracoes++;
        }

        content += "const unsigned int event[NTRANS]={"+ events +"} \n";
        content += "const unsigned int in_state[NTRANS]={"+ in_states +"} \n";
        content += "const unsigned int rfirst[NESTADOS]={"+ rfirst +"} \n";
        content += "const unsigned int rnext[NTRANS]={"+ rnext +"} \n\n";

        writeContent(content);
}
```

Listagem 5 – Método de definição dos valores dos vetores

Já para os *arrays* de *rfirst* e *rnext*, é necessário fazer algumas verificações antes. Com a função *contem*, é possível verificar se o elemento já existe no vetor de transições, e caso sim retorna à posição do último estado possível de acontecer. Para *rnext*, a implementação é parecida com a do *rfirst*, só que a diferença é que a função *contem*, retorna a próxima posição do estado atual.

Na Listagem 6, apresenta-se o método *main*. Inicialmente tem-se a linha que desabilita o WDT, as configurações de *clock*, timer e entradas e saídas. Na sequência aparece algumas variáveis locais:

a) *occur_event* que representa o estado ocorrido;
b) *current_state* representa o estado atual inicializado com o estado inicial;
c) g é a *flag* geradora aleatória de eventos;
d) *gerar_evento* é a *flag* que habilita a temporização de eventos controláveis;
e) *mealy_output* que inicia a saída periférica;
f) *_enable_interrupt* que habilita a interrupção geral;

No *loop while*, existe três blocos de código *if else*, o primeiro serve para quando não existir evento no *buffer* então gera um evento interno, um evento não controlável. O segundo bloco é o gerador do Autômato e o terceiro bloco mostra que se o evento ocorrido não for válido, então imprime a saída física. Neste exemplo a saída é referente a Máquina de Mealy, portanto o *switch* case executa de acordo com o evento ocorrido, mas caso fosse uma saída referente a Máquina de Moore, então o *switch* case executaria de acordo com o estado ocorrido.

```
private void setMain(){

	content += "void main(void)";
	content += "{ \n";

	content += "   WDTCTL = WDTPW + WDTHOLD;\n";
	content += "   config_clk();\n";
	content += "   config_timer();\n";
	content += "   config_io();\n\n";

	content += "   unsigned int k;\n";
	content += "   int occur_event;\n";
	content += "   unsigned int current_state = 0;\n";
	content += "   int g=0;\n";
	content += "   int gerar_evento=1;\n";
	content += "   int mealy_output = 0;\n";
	content += "   _enable_interrupt();\n\n";

	content += "   while(1)\n ";
	content += "   {\n\n";

	content += "           if(n_buffer == 0)\n";
	content += "           {\n";
	content += "                   if(TACTL&TAIFG)\n";
	content += "                   {\n";
	content += "                           gerar_evento=1;\n";
	content += "                   }\n";
	content += "                   if(gerar_evento==1)\n";
	content += "                   {\n";
	content += "                           switch(g)\n";
	content += "                           {\n";

	int count = 0;
	String g= "g++";
	for(String evc : vectorEvtControl){

                        if (vectorEvtControl.size()-1 ==count) {
                        g = "g=0";
                }
                content += "                                   case(" + count +"):\n";
                content += "                                           occur_event="  +  evc  +  ";
\n";
                content += "                                           " + g +";\n";
                content += "                                           break;\n";

                count++;
	}

	content += "                           }\n";
	content += "                   }\n";
	content += "           }\n";
	content += "           else\n";
	content += "           {\n";
	content += "                   occur_event = buffer[0];\n";
	content += "                   n_buffer--;\n";
	content += "                   k = 0;\n";
	content += "                   while(k<n_buffer)\n";
	content += "                   {\n";
	content += "                           buffer[k] = buffer[k+1];\n";
	content += "                           k++;\n";
```

```
        content += "                    }\n";
        content += "           }\n\n";

        content += "           k = rfirst[current_state];\n";
        content += "           if(k==0)\n";
        content += "           {\n";
        content += "                   return;     //Dead Lock!!!\n";
        content += "           }\n";
        content += "           else\n";
        content += "           {\n";
        content += "                   while(k>0)\n";
        content += "                   {\n ";
        content += "                           k--;\n";
        content += "                           if(event[k] == occur_event)\n";
        content += "                           {\n";
        content += "                                   current_state = in_state[k];\n";
        content += "                                   mealy_output = 1;\n";
        content += "                                   break;\n";
        content += "                           }\n";
        content += "                           k = rnext[k];\n";
        content += "                   }\n";
        content += "           }\n";

        content += "           if(mealy_output)\n";
        content += "           {\n";
        content += "                   switch(occur_event)\n";
        content += "                   {\n";
        content += "                           //eventos controláveis\n";

        for(String ec : vectorEvtControl){
                content += "                           case(" + ec + "):\n\n";
                content += "                                   break;\n";

        }

        for(String enc : vectorEvtNaoControl){
                content += "                           case(" + enc + "):\n\n";
                content += "                                   gerar_evento=1;\n";
                content += "                                   break;\n";

        }

        content += "                   }//fim switch\n";
        content += "                   mealy_output = 0;\n";
        content += "                   occur_event = -1;\n";
        content += "           }//fim if(mealy_output)\n";
        content += "   }//fim while(1)";
        content += "}//fim main\n\n\n";

}
```

Listagem 6 – Método principal - *Main*

E por fim, na Listagem 7, é apresentado os métodos destinados ao microcontrolador. São os métodos: *__interrupt* que faz o tratamento da interrupção da porta P1, o *config_clk* que configura o *clock*, o *config_timer* que configura o timer A, o *config_io* que configura as entradas e saídas e o *tempo_ms* que é a função de temporização do *timer* A.

```
//Tratamento da Interrupção da porta P1
//-------------------------------------------------------------
#pragma vector=PORT1_VECTOR
__interrupt void RTI_PORT1(void)
{
        if(P1IFG&BIT0)
        {
                P1IFG&=~BIT0;
                buffer[n_buffer]=E0;
```

```
                if(n_buffer<BUFFER-1) n_buffer++;
        }
        P1IFG=0;
}

//Configuração de perifericos
//-------------------------------------------------------------
void config_clk(void)
{
        DCOCTL = 0x00;
        __delay_cycles(100000);
        DCOCTL = MOD0 + MOD1 + MOD2 + MOD3 + MOD4 + DCO0;
        BCSCTL1 = XT2OFF + DIVA_0 + RSEL1 + RSEL2 + RSEL3;
}

//Configura o timer
//-------------------------------------------------------------
void config_timer(void)
{
        TACTL = TASSEL_1 + ID_2 + MC_1;
        BCSCTL3 |= LFXT1S1;
}

//configura entradas e saídas
//-------------------------------------------------------------
void config_io(void)
{
        P1DIR = 0;              //toda a porta P1 como entrada
        P1REN |= BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6+BIT7;
        P1OUT |= BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6+BIT7;
        P1IE  |= BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6+BIT7;
        P1IES |= BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6+BIT7;
        P1IFG = 0;
        P2SEL &= ~(BIT6 | BIT7);
        P2DIR |= BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6+BIT7;
        P2OUT = 0;
}

//Função de temporização do timer A
//-------------------------------------------------------------
void tempo_ms(unsigned int tempo)
{
        TACTL|= TACLR;
        TACCR0 =3*tempo;
        TACTL &= ~TAIFG;
}
```

Listagem 7 – Métodos de configuração do microcontrolador

Para finalizar, a aplicação recebeu o nome de ConverterC. E quando executada, uma tela será exibida para que o usuário informe o nome do arquivo XML e o seu local correspondente e após isso, clicar em OK.

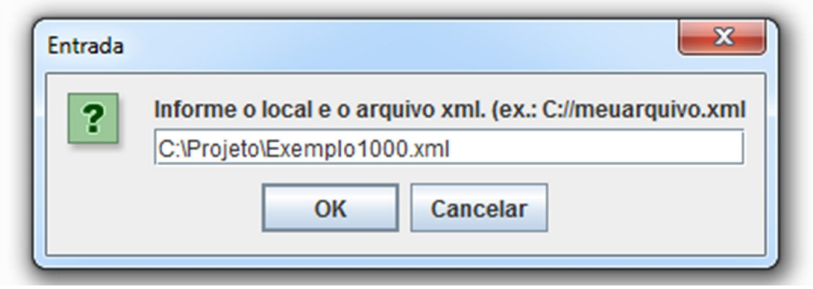

Figura 8 - Entrada da aplicação

A aplicação vai criar uma pasta denominada “ConverterC” no disco local e dentro desta pasta irá criar um arquivo chamado de automato.c.

Neste arquivo estará o código na linguagem C, como mostrado na Listagem 8, que será executado pelo microcontrolador MSP430.

```
#include "msp430.h"

//Dados do autômato
#define NTRANS 4
#define NESTADOS 2
#define BUFFER 10

const unsigned int event[NTRANS]={0, 1, 2, 3}
const unsigned int in_state[NTRANS]={1, 0, 0, 1}
const unsigned int rfirst[NESTADOS]={2, 4}
const unsigned int rnext[NTRANS]={0, 1, 0, 3}

#define E0 0
#define E1 2

 #define S0_ON  P2OUT |= BIT0 //Saida 0 ON
 #define S0_OFF P2OUT &=~BIT0 //Saida 0 OFF
 #define S1_ON  P2OUT |= BIT1 //Saida 1 ON
 #define S1_OFF P2OUT &=~BIT1 //Saida 1 OFF
 #define S2_ON  P2OUT |= BIT2 //Saida 2 ON
 #define S2_OFF P2OUT &=~BIT2 //Saida 2 OFF
 #define S3_ON  P2OUT |= BIT3 //Saida 3 ON
 #define S3_OFF P2OUT &=~BIT3 //Saida 3 OFF
 #define S4_ON  P2OUT |= BIT4 //Saida 4 ON
 #define S4_OFF P2OUT &=~BIT4 //Saida 4 OFF
 #define S5_ON  P2OUT |= BIT5 //Saida 5 ON
 #define S5_OFF P2OUT &=~BIT5 //Saida 5 OFF
 #define S6_ON  P2OUT |= BIT6 //Saida 6 ON
 #define S6_OFF P2OUT &=~BIT6 //Saida 6 OFF
 #define S7_ON  P2OUT |= BIT7 //Saida 7 ON
 #define S7_OFF P2OUT &=~BIT7 //Saida 7 OFF

void config_clk(void);
void config_timer(void);
void config_io(void);
void tempo_ms(unsigned int);

unsigned int buffer[BUFFER];
unsigned int n_buffer=0;

void main(void)
{
	WDTCTL = WDTPW + WDTHOLD;
	config_clk();
	config_timer();
	config_io();

	unsigned int k;
	int occur_event;
	unsigned int current_state = 0;
	int g=0;
	int gerar_evento=1;
	int mealy_output = 0;
	_enable_interrupt();

	while(1)
	{

		if(n_buffer == 0)
		{
			if(TACTL&TAIFG)
			{
				gerar_evento=1;
			}
```

```
                        if(gerar_evento==1)
                        {
                                switch(g)
                                {
                                        case(0):
                                                occur_event=1;
                                                g++;
                                                break;
                                        case(1):
                                                occur_event=3;
                                                g=0;
                                                break;
                                }
                        }
                }
                else
                {
                        occur_event = buffer[0];
                        n_buffer--;
                        k = 0;
                        while(k<n_buffer)
                        {
                                buffer[k] = buffer[k+1];
                                k++;
                        }
                }

                k = rfirst[current_state];
                if(k==0)
                {
                        return;     //Dead Lock!!!
                }
                else
                {
                        while(k>0)
                        {
                                k--;
                                if(event[k] == occur_event)
                                {
                                        current_state = in_state[k];
                                        mealy_output = 1;
                                        break;
                                }
                                k = rnext[k];
                        }
                }
                if(mealy_output)
                {
                        switch(occur_event)
                        {
                                //eventos controláveis
                                case(1):

                                        break;
                                case(3):

                                        break;
                                case(0):

                                        gerar_evento=1;
                                        break;
                                case(2):

                                        gerar_evento=1;
                                        break;
                        }//fim switch
                        mealy_output = 0;
                        occur_event = -1;
                }//fim if(mealy_output)
        }//fim while(1)}//fim main


 #pragma vector=PORT1_VECTOR
__interrupt void RTI_PORT1(void)
{
        if(P1IFG&BIT0)
        {
```

```
                P1IFG&=~BIT0;
                buffer[n_buffer]=E0;
                if(n_buffer<BUFFER-1) n_buffer++;
        }
        if(P1IFG&BIT1)
        {
                P1IFG&=~BIT1;
                buffer[n_buffer]=E1;
                if(n_buffer<BUFFER-1) n_buffer++;
        }
        P1IFG=0;
}

 void config_clk(void)
{
        DCOCTL = 0x00;
        __delay_cycles(100000);
        DCOCTL = MOD0 + MOD1 + MOD2 + MOD3 + MOD4 + DCO0;
        BCSCTL1 = XT2OFF + DIVA_0 + RSEL1 + RSEL2 + RSEL3;
}
 void config_timer(void)
{
        TACTL = TASSEL_1 + ID_2 + MC_1;
        BCSCTL3 |= LFXT1S1;
}
 void config_timer(void)
{
        P1DIR = 0;
        P1REN |= BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6+BIT7;
        P1OUT |= BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6+BIT7;
        P1IE  |= BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6+BIT7;
        P1IES |= BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6+BIT7;
        P1IFG = 0;
        P2SEL &= ~(BIT6 | BIT7);
        P2DIR |= BIT0+BIT1+BIT2+BIT3+BIT4+BIT5+BIT6+BIT7;
        P2OUT = 0;
}
 void tempo_ms(unsigned int tempo)
{
        TACTL|= TACLR;
        TACCR0 =3*tempo;
        TACTL &= ~TAIFG;
}
```

Listagem 8 – Código de saída da aplicação

Com a aplicação proposta, um Autômato que representa o controle de um processo pode ser exportado diretamente do Supremica para o microcontrolador. Entende-se que essa facilidade contribui para aproximar a prática industrial às ferramentas de modelagem e aos métodos formais.

6 CONCLUSÃO

O trabalho abordou sobre como integrar o Supremica com os microcontroladores MSP430, da *Texas Instruments*. O Supremica utiliza modelagem de máquinas de estados e permite analisar, sintetizar e simular o software por meio de máquinas de estados. Este tipo de modelagem se faz importante, principalmente, quando precisa especificar inúmeros sistemas e requisitos e quando se pretende obter redução de custos e eliminação de erros.

As vantagens da aplicação desenvolvida é a possibilidade de otimizar tempo e trabalho ao criar o modelo no Supremica e poder obter o modelo na linguagem C.

Algumas dificuldades foram encontradas no decorrer do trabalho, como a assimilação da estrutura do modelo de Autômatos entre o XML e o código C.

Os objetivos foram alcançados e ara trabalhos futuros, ou para até mesmo profissionais da área com conhecimento mais aprofundado, algumas melhorias podem ser realizadas quanto a técnica de desenvolvimento, aplicação de técnicas mais avançadas como por exemplo, compiladores, ou ainda integrar a aplicação dentro do Supremica, otimizando mais ainda o processo, na forma de *plug-in*.

REFERÊNCIAS

AKESSON, Knut; MARTIN, Fabian; FLORDAL, Hugo; VAHIDI, Arash. Supremica – *a tool for verification and synthesis of discrete event supervisors***,** Göteborg, p.1-6, jun. 2006.

CASSANDRAS, C. G; LAFORTUNE, S. ***Introduction to discrete event systems***. 2 ed., Springer, 2008.

MENEZES, Paulo B. **Microcontroladores MSP430**: Teoria e Prática. São Paulo: Érica, 2000.

PEREIRA, Fabio. **Linguagens Formais e Autômatos**. Porto Alegre: Sagra, 2005.

RAMADGE P.; WONHAM, W. ***The control of discrete event systems.*** *In: Proceendings ieee – special issue on discrete event dynamic systems,* v.77, 1989.

Texas Instruments. *Getting started for MSP430 ultra-low-power MCUs*. 2016. Disponível em: <http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/getting-started.page#step1>. Acesso em: 21 jan. 2017.